Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8

By Tom Dykstra, Jeremy Likness, and Jon P Smith

This is the first in a series of tutorials that show how to use Entity Framework (EF) Core in an ASP.NET Core Razor Pages app. The tutorials build a web site for a fictional Contoso University. The site includes functionality such as student admission, course creation, and instructor assignments. The tutorial uses the code first approach. For information on following this tutorial using the database first approach, see this Github issue.

Download or view the completed app. Download instructions.

Prerequisites

- If you're new to Razor Pages, go through the Get started with Razor Pages tutorial series before starting this one.
- Visual Studio
- Visual Studio Code
- Visual Studio 2022 with the **ASP.NET and web development** workload.

Database engines

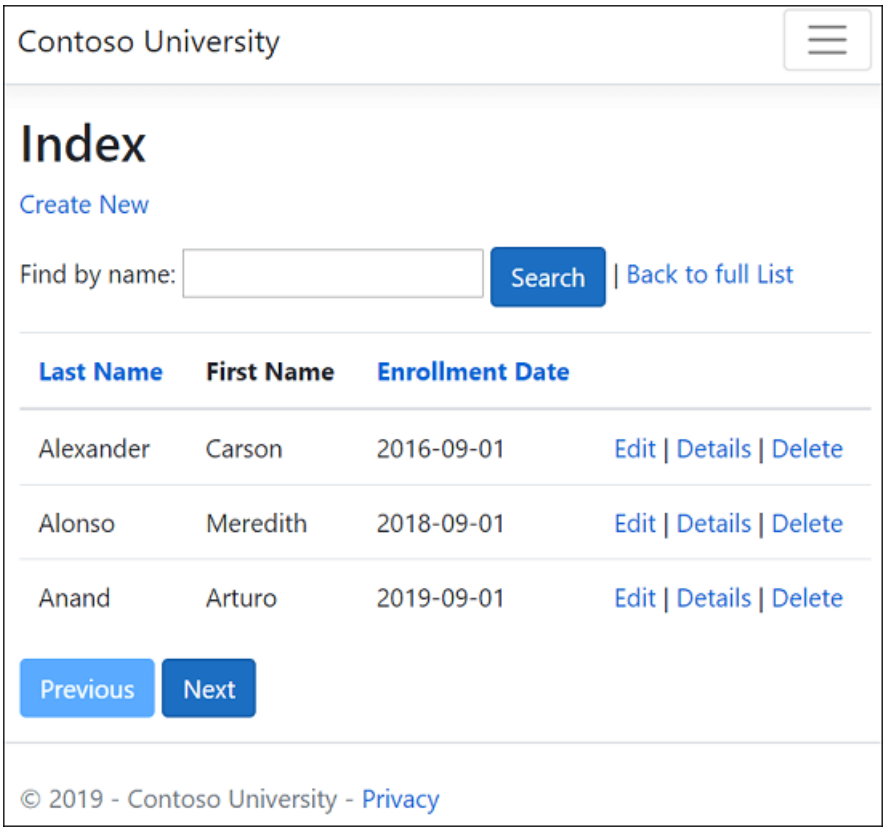
The Visual Studio instructions use SQL Server LocalDB, a version of SQL Server Express that runs only on Windows.

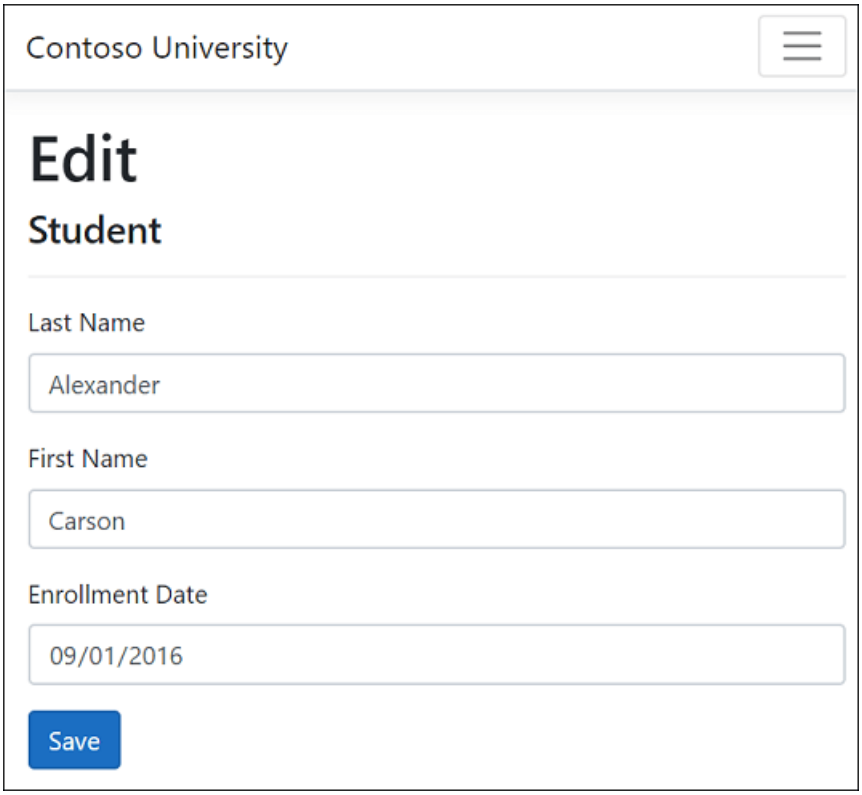
Troubleshooting

If you run into a problem you can't resolve, compare your code to the completed project. A good way to get help is by posting a question to StackOverflow.com, using the ASP.NET Core tag or the EF Core tag.

The sample app

The app built in these tutorials is a basic university web site. Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.





The UI style of this site is based on the built-in project templates. The tutorial's focus is on how to use EF Core with ASP.NET Core, not how to customize the UI.

Optional: Build the sample download

This step is optional. Building the completed app is recommended when you have problems you can't solve. If you run into a problem you can't resolve, compare your code to the completed project. Download instructions.

- Visual Studio
- Visual Studio Code

Select `ContosoUniversity.csproj` to open the project.

- Build the project.
- In Package Manager Console (PMC) run the following command:

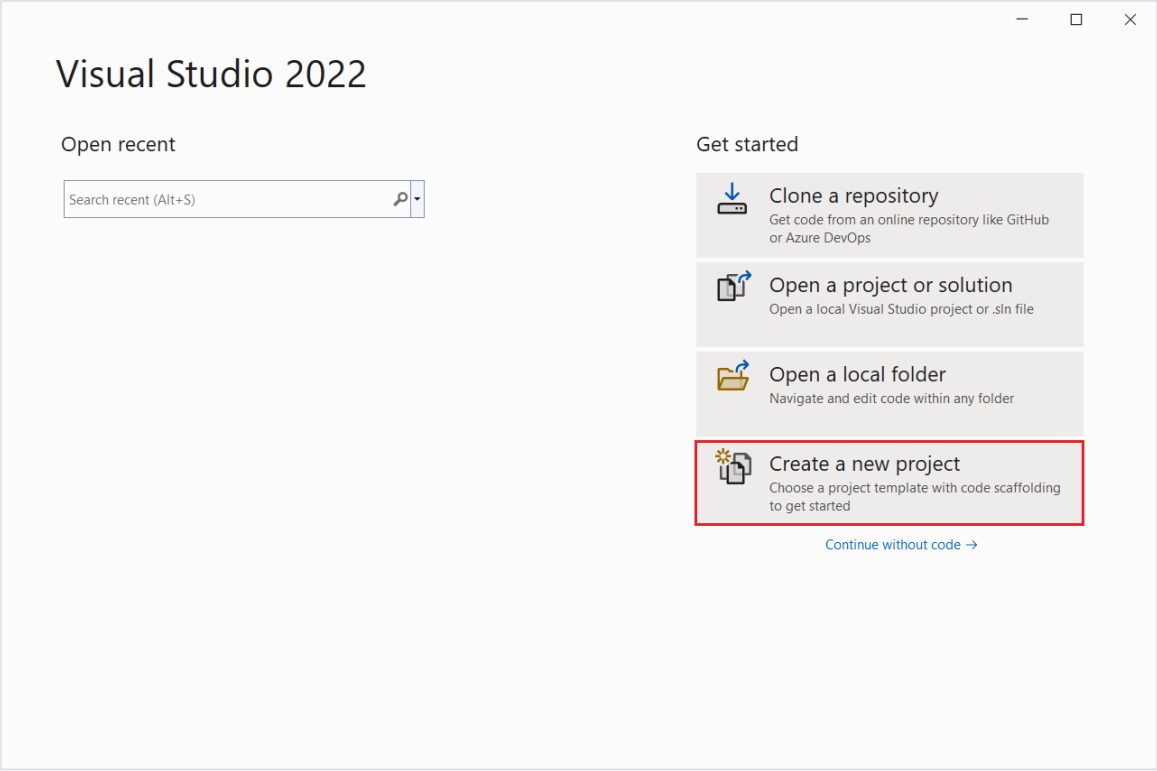
```
Update-Database
```

Run the project to seed the database.

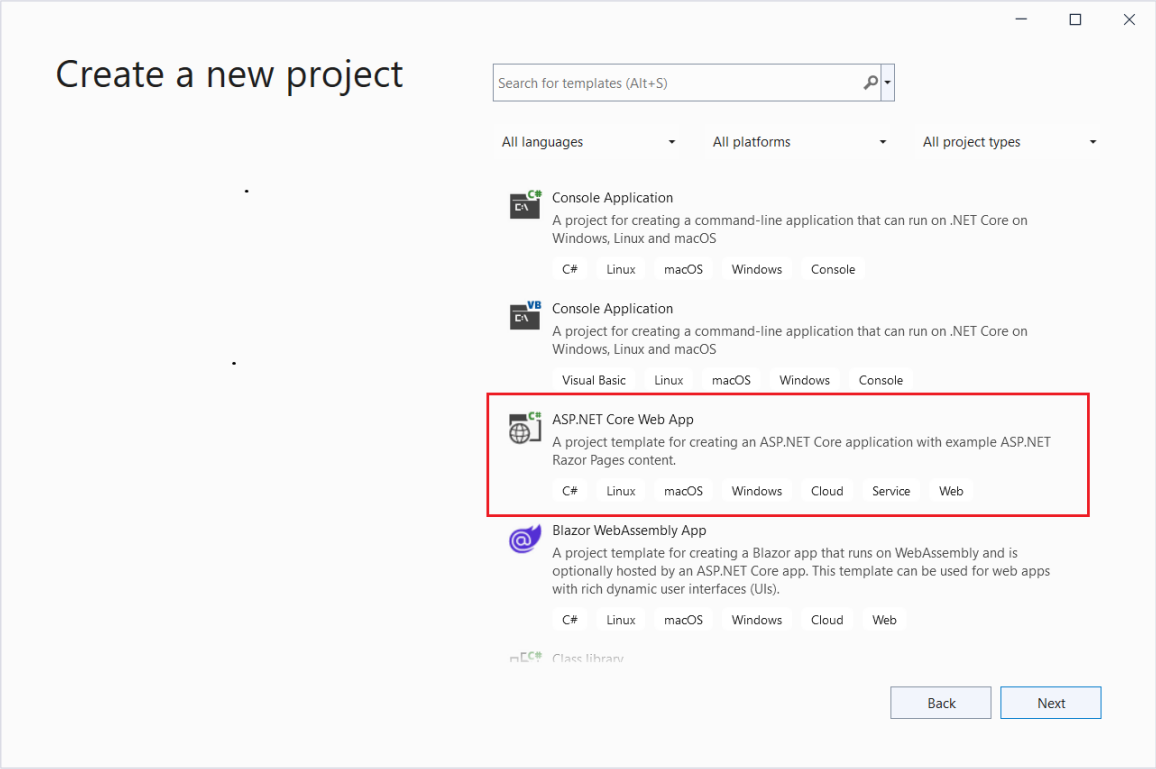
Create the web app project

- Visual Studio
- Visual Studio Code

1. Start Visual Studio 2022 and select **Create a new project**.



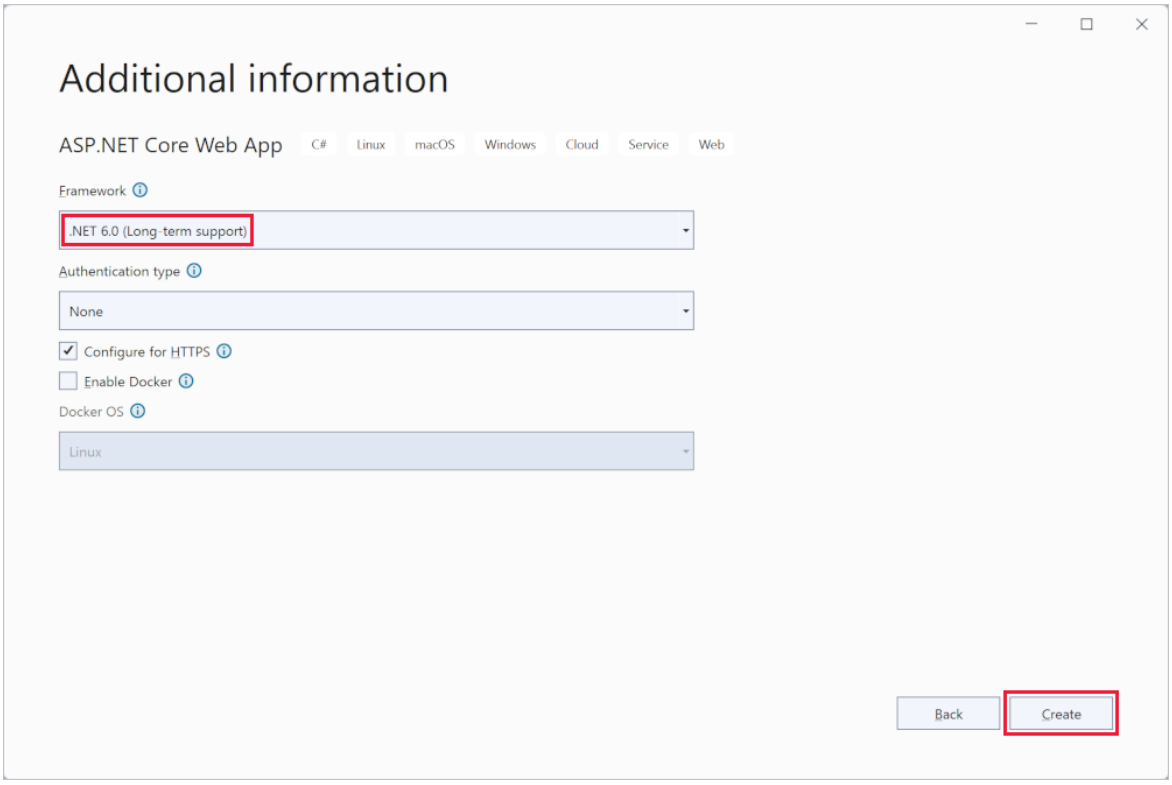
2. In the **Create a new project** dialog, select **ASP.NET Core Web App**, and then select **Next**.



3. In the **Configure your new project** dialog, enter `ContosoUniversity` for **Project name**. It's important to name the project *ContosoUniversity*, including matching the capitalization, so the namespaces will match when you copy and paste example code.

4. Select **Next**.

5. In the **Additional information** dialog, select **.NET 6.0 (Long-term support)** and then select **Create**.



Set up the site style

Copy and paste the following code into the `Pages/Shared/_Layout.cshtml` file:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
    <link rel="stylesheet" href="~/ContosoUniversity.styles.css" asp-append-version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-
shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-page="/Index">Contoso University</a>
                <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-
target=".navbar-collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Students/Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/Courses/Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Instructors/Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Departments/Index">Departments</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2021 - Contoso University - <a asp-area="" asp-page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

The layout file sets the site header, footer, and menu. The preceding code makes the following changes:

- Each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- The **Home** and **Privacy** menu entries are deleted.
- Entries are added for **About**, **Students**, **Courses**, **Instructors**, and **Departments**.

In `Pages/Index.cshtml`, replace the contents of the file with the following code:

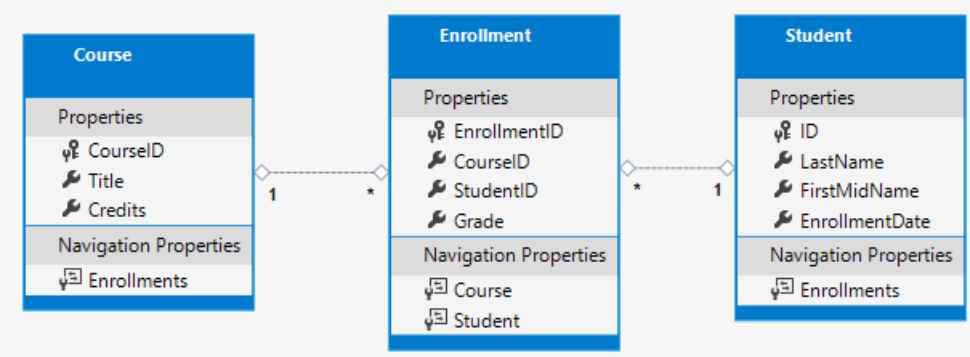
```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="row mb-auto">
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 mb-4 ">
                <p class="card-text">
                    Contoso University is a sample application that
                    demonstrates how to use Entity Framework Core in an
                    ASP.NET Core Razor Pages web app.
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column position-static">
                <p class="card-text mb-auto">
                    You can build the application by following the steps in a series of tutorials.
                </p>
                <p>
@*                    <a href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro" class="stretched-
link">See the tutorial</a>
*@                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column">
                <p class="card-text mb-auto">
                    You can download the completed project from GitHub.
                </p>
                <p>
@*                    <a href="https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/data/ef-
rp/intro/samples" class="stretched-link">See project source code</a>
*@                </p>
            </div>
        </div>
    </div>
</div>
```

The preceding code replaces the text about ASP.NET Core with text about this app.

Run the app to verify that the home page appears.

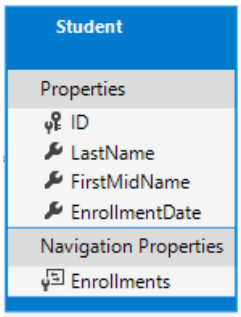
The data model

The following sections create a data model:



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

The Student entity



- Create a *Models* folder in the project folder.
- Create `Models/Student.cs` with the following code:

```
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; }
    }
}
```

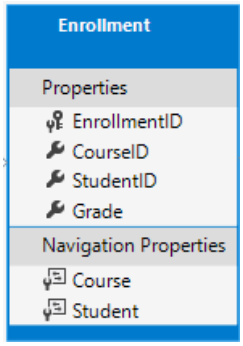
The `ID` property becomes the primary key column of the database table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key. So the alternative automatically recognized name for the `Student` class primary key is `StudentID`. For more information, see EF Core - Keys.

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 holds all of the `Enrollment` entities that are related to that Student. For example, if a Student row in the database has two related Enrollment rows, the `Enrollments` navigation property contains those two Enrollment entities.

In the database, an Enrollment row is related to a Student row if its `StudentID` column contains the student's ID value. For example, suppose a Student row has ID=1. Related Enrollment rows will have `StudentID` = 1. `StudentID` is a *foreign key* in the Enrollment table.

The `Enrollments` property is defined as `ICollection<Enrollment>` because there may be multiple related Enrollment entities. Other collection types can be used, such as `List<Enrollment>` or `HashSet<Enrollment>`. When `ICollection<Enrollment>` is used, EF Core creates a `HashSet<Enrollment>` collection by default.

The Enrollment entity



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

```
using System.ComponentModel.DataAnnotations;

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; }
    }
}
```

The `EnrollmentID` property is the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself. For a production data model, many developers choose one pattern and use it consistently. This tutorial uses both just to illustrate that both work. Using `ID` without `classname` makes it easier to implement some kinds of data model changes.

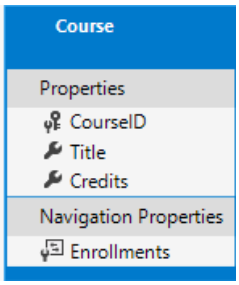
The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade—null means a grade isn't known or hasn't been assigned yet.

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 contains a single `Student` entity.

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

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

The Course entity



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

```
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.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the database generate it.

Build the app. The compiler generates several warnings about how `null` values are handled. See this GitHub issue, Nullable reference types, and Tutorial: Express your design intent more clearly with nullable and non-nullable reference types for more information.

To eliminate the warnings from nullable reference types, remove the following line from the `ContosoUniversity.csproj` file:

```
<Nullable>enable</Nullable>
```

The scaffolding engine currently does not support nullable reference types, therefore the models used in scaffold can't either.

Remove the `?` nullable reference type annotation from `public string? RequestId { get; set; }` in `Pages/Error.cshtml.cs` so the project builds without compiler warnings.

Scaffold Student pages

In this section, the ASP.NET Core scaffolding tool is used to generate:

- An EF Core `DbContext` class. The context is the main class that coordinates Entity Framework functionality for a given data model. It derives from the Microsoft.EntityFrameworkCore.DbContext class.
- Razor pages that handle Create, Read, Update, and Delete (CRUD) operations for the `Student` entity.

- Visual Studio
- Visual Studio Code

- Create a *Pages/Students* folder.
- In **Solution Explorer**, right-click the *Pages/Students* folder and select **Add** > **New Scaffolded Item**.
- In the **Add New Scaffold Item** dialog:
  - In the left tab, select **Installed** > **Common** > **Razor Pages**
  - Select **Razor Pages using Entity Framework (CRUD)** > **ADD**.
- In the **Add Razor Pages using Entity Framework (CRUD)** dialog:
  - In the **Model class** drop-down, select **Student (ContosoUniversity.Models)**.
  - In the **Data context class** row, select the **+** (plus) sign.
    - Change the data context name to end in `SchoolContext` rather than `ContosoUniversityContext`.
      The updated context name: `ContosoUniversity.Data.SchoolContext`
    - Select **Add** to finish adding the data context class.
    - Select **Add** to finish the **Add Razor Pages** dialog.

The following packages are automatically installed:

- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.EntityFrameworkCore.Tools`
- `Microsoft.VisualStudio.Web.CodeGeneration.Design`

If the preceding step fails, build the project and retry the scaffold step.

The scaffolding process:

- Creates Razor pages in the *Pages/Students* folder:
  - `Create.cshtml` and `Create.cshtml.cs`
  - `Delete.cshtml` and `Delete.cshtml.cs`
  - `Details.cshtml` and `Details.cshtml.cs`
  - `Edit.cshtml` and `Edit.cshtml.cs`
  - `Index.cshtml` and `Index.cshtml.cs`
- Creates `Data/SchoolContext.cs`.
- Adds the context to dependency injection in `Program.cs`.
- Adds a database connection string to `appsettings.json`.

Database connection string

The scaffolding tool generates a connection string in the `appsettings.json` file.

- Visual Studio
- Visual Studio Code

The connection string specifies SQL Server LocalDB:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=(localdb)\\mssqllocaldb;Database=SchoolContext-
0e9;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. By default, LocalDB creates *.mdf* files in the `C:/Users/<user>` directory.

Update the database context class

The main class that coordinates EF Core functionality for a given data model is the database context class. The context is derived from Microsoft.EntityFrameworkCore.DbContext. The context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

Update `Data/SchoolContext.cs` with the following code:

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

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

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

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

The preceding code changes from the singular `DbSet<Student> Student` to the plural `DbSet<Student> Students`. To make the Razor Pages code match the new `DBSet` name, make a global change from: `_context.Student.` to: `_context.Students.`

There are 8 occurrences.

Because an entity set contains multiple entities, many developers prefer the `DBSet` property names should be plural.

The highlighted code:

- Creates a DbSet<TEntity> property for each entity set. In EF Core terminology:
  - An entity set typically corresponds to a database table.
  - An entity corresponds to a row in the table.
- Calls OnModelCreating. `OnModelCreating`:
  - Is called when `SchoolContext` has been initialized, but before the model has been locked down and used to initialize the context.
  - Is required because later in the tutorial the `Student` entity will have references to the other entities.

We hope to fix this issue in a future release.

Program.cs

ASP.NET Core is built with dependency injection. Services such as the `SchoolContext` are registered with dependency injection during app startup. Components that require these services, such as Razor Pages, are provided these services via constructor parameters. The constructor code that gets a database context instance is shown later in the tutorial.

The scaffolding tool automatically registered the context class with the dependency injection container.

- Visual Studio
- Visual Studio Code

The following highlighted lines were added by the scaffolder:

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDbContext<SchoolContext>(options =>
  options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolContext")));
```

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the `appsettings.json` or the `appsettings.Development.json` file.

Add the database exception filter

Add AddDatabaseDeveloperPageExceptionFilter and UseMigrationsEndPoint as shown in the following code:

- Visual Studio
- Visual Studio Code

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDbContext<SchoolContext>(options =>
  options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolContext")));

builder.Services.AddDatabaseDeveloperPageExceptionFilter();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
else
{
    app.UseDeveloperExceptionPage();
    app.UseMigrationsEndPoint();
}
```

Add the Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore NuGet package.

In the Package Manager Console, enter the following to add the NuGet package:

```
Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
```

The `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore` NuGet package provides ASP.NET Core middleware for Entity Framework Core error pages. This middleware helps to detect and diagnose errors with Entity Framework Core migrations.

The `AddDatabaseDeveloperPageExceptionFilter` provides helpful error information in the development environment for EF migrations errors.

Create the database

Update `Program.cs` to create the database if it doesn't exist:

- Visual Studio
- Visual Studio Code

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDbContext<SchoolContext>(options =>
  options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolContext")));

builder.Services.AddDatabaseDeveloperPageExceptionFilter();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
else
{
    app.UseDeveloperExceptionPage();
    app.UseMigrationsEndPoint();
}

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    var context = services.GetRequiredService<SchoolContext>();
    context.Database.EnsureCreated();
    // DbInitializer.Initialize(context);
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The EnsureCreated method takes no action if a database for the context exists. If no database exists, it creates the database and schema. `EnsureCreated` enables the following workflow for handling data model changes:

- Delete the database. Any existing data is lost.
- Change the data model. For example, add an `EmailAddress` field.
- Run the app.
- `EnsureCreated` creates a database with the new schema.

This workflow works early in development when the schema is rapidly evolving, as long as data doesn't need to be preserved. The situation is different when data that has been entered into the database needs to be preserved. When that is the case, use migrations.

Later in the tutorial series, the database is deleted that was created by `EnsureCreated` and migrations is used. A database that is created by `EnsureCreated` can't be updated by using migrations.

Test the app

- Run the app.
- Select the **Students** link and then **Create New**.

- Test the Edit, Details, and Delete links.

Seed the database

The `EnsureCreated` method creates an empty database. This section adds code that populates the database with test data.

Create `Data/DbInitializer.cs` with the following code:

```
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            // 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("2019-
09-01")},
                new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2017-
09-01")},
                new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2018-09-
01")},
                new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2017-
09-01")},
                new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2017-09-01")},
                new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2016-09-
01")},
                new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2018-09-
01")},
                new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2019-09-
01")}
            };

            context.Students.AddRange(students);
            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}
            };

            context.Courses.AddRange(courses);
            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},
            };

            context.Enrollments.AddRange(enrollments);
            context.SaveChanges();
        }
    }
}
```

The code checks if there are any students in the database. If there are no students, it adds test data to the database. It creates the test data in arrays rather than `List<T>` collections to optimize performance.

- In `Program.cs`, remove `//` from the `DbInitializer.Initialize` line:

```
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    var context = services.GetRequiredService<SchoolContext>();
    context.Database.EnsureCreated();
    DbInitializer.Initialize(context);
}
```

- Visual Studio
- Visual Studio Code

- Stop the app if it's running, and run the following command in the **Package Manager Console** (PMC):

```
Drop-Database -Confirm
```

- Respond with `Y` to delete the database.

- Restart the app.
- Select the Students page to see the seeded data.

View the database

- Visual Studio
- Visual Studio Code

- Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio.
- In SSOX, select **(localdb)\MSSQLLocalDB > Databases > SchoolContext-{GUID}**. The database name is generated from the context name provided earlier plus a dash and a GUID.
- Expand the **Tables** node.
- Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into the table.
- Right-click the **Student** table and click **View Code** to see how the `Student` model maps to the `Student` table schema.

Asynchronous EF methods in ASP.NET Core web apps

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 doing 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 can handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. 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` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task OnGetAsync()
{
    Students = await _context.Students.ToListAsync();
}
```

- The `async` keyword tells the compiler to:
  - Generate callbacks for parts of the method body.
  - Create the Task object that's returned.
- The `Task` return type represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF Core context isn't thread safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the database.

For more information about asynchronous programming in .NET, see Async Overview and Asynchronous programming with async and await.

Performance considerations

In general, a web page shouldn't be loading an arbitrary number of rows. A query should use paging or a limiting approach. For example, the preceding query could use `Take` to limit the rows returned:

```
public async Task OnGetAsync()
{
    Student = await _context.Students.Take(10).ToListAsync();
}
```

Enumerating a large table in a view could return a partially constructed HTTP 200 response if a database exception occurs part way through the enumeration.

Paging is covered later in the tutorial.

For more information, see Performance considerations (EF).

Next steps

Use SQLite for development, SQL Server for production

NEXT TUTORIAL

This is the first in a series of tutorials that show how to use Entity Framework (EF) Core in an ASP.NET Core Razor Pages app. The tutorials build a web site for a fictional Contoso University. The site includes functionality such as student admission, course creation, and instructor assignments. The tutorial uses the code first approach. For information on following this tutorial using the database first approach, see this Github issue.

Download or view the completed app. Download instructions.

Prerequisites

- If you're new to Razor Pages, go through the Get started with Razor Pages tutorial series before starting this one.

- Visual Studio
- Visual Studio Code

- Visual Studio 2019 16.8 or later with the **ASP.NET and web development** workload
- .NET 5.0 SDK

Database engines

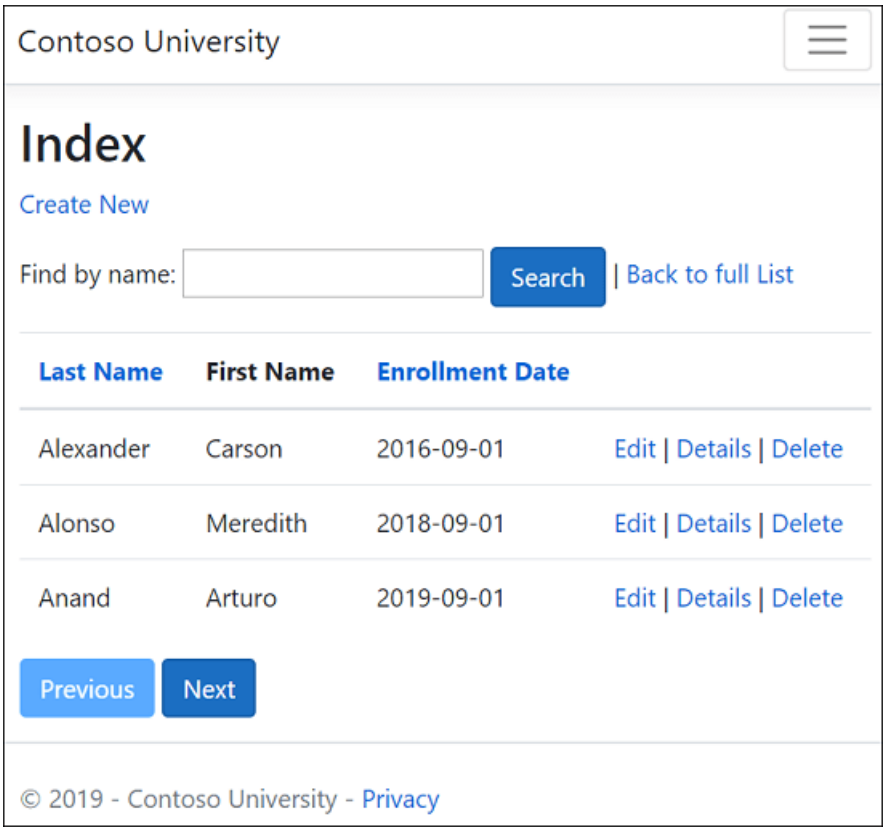
The Visual Studio instructions use SQL Server LocalDB, a version of SQL Server Express that runs only on Windows.

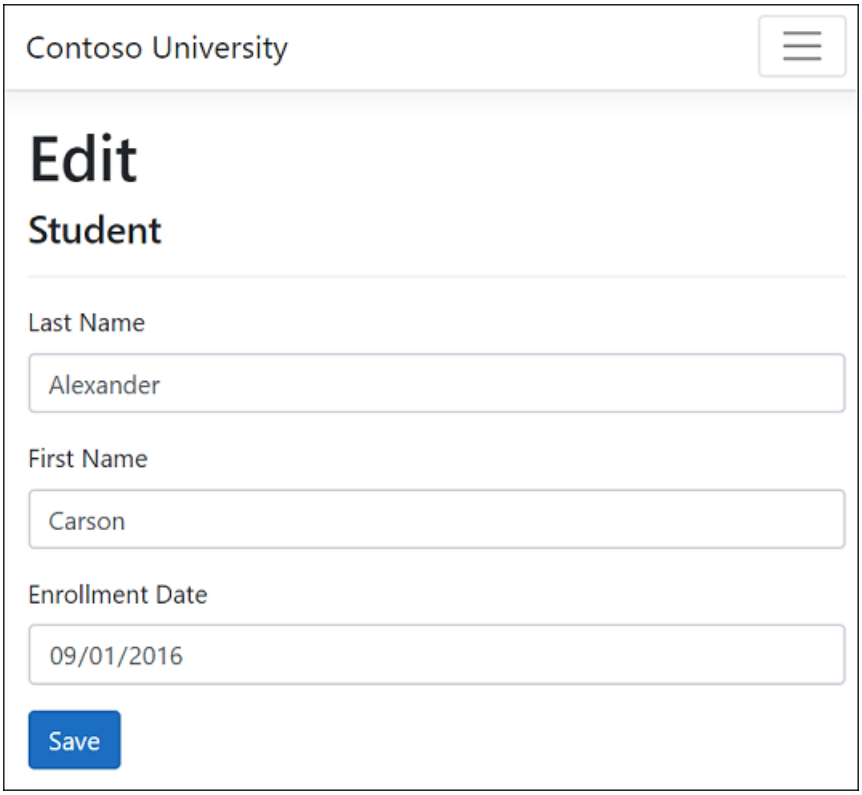
Troubleshooting

If you run into a problem you can't resolve, compare your code to the completed project. A good way to get help is by posting a question to StackOverflow.com, using the ASP.NET Core tag or the EF Core tag.

The sample app

The app built in these tutorials is a basic university web site. Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.





The UI style of this site is based on the built-in project templates. The tutorial's focus is on how to use EF Core with ASP.NET Core, not how to customize the UI.

Optional: Build the sample download

This step is optional. Building the completed app is recommended when you have problems you can't solve. If you run into a problem you can't resolve, compare your code to the completed project. Download instructions.

- Visual Studio
- Visual Studio Code

Select `ContosoUniversity.csproj` to open the project.

- Build the project.
- In Package Manager Console (PMC) run the following command:

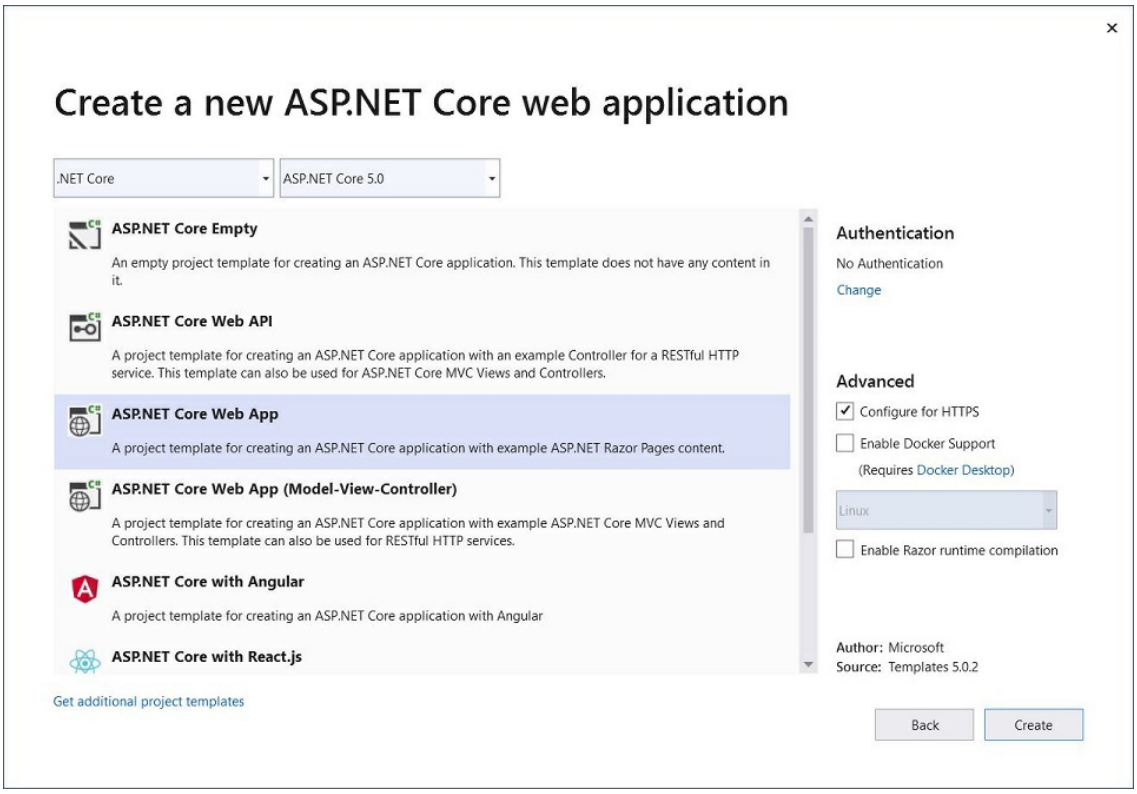
```
Update-Database
```

Run the project to seed the database.

Create the web app project

- Visual Studio
- Visual Studio Code

1. Start Visual Studio and select **Create a new project**.
2. In the **Create a new project** dialog, select **ASP.NET Core Web Application** > **Next**.
3. In the **Configure your new project** dialog, enter `ContosoUniversity` for **Project name**. It's important to use this exact name including capitalization, so each `namespace` matches when code is copied.
4. Select **Create**.
5. In the **Create a new ASP.NET Core web application** dialog, select:
   a. **.NET Core** and **ASP.NET Core 5.0** in the dropdowns.
   b. **ASP.NET Core Web App**.
   c. **Create**



Set up the site style

Copy and paste the following code into the `Pages/Shared/_Layout.cshtml` file:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-
shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-page="/Index">Contoso University</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-
collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Students/Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/Courses/Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Instructors/Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Departments/Index">Departments</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2021 - Contoso University - <a asp-area="" asp-page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

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

The layout file sets the site header, footer, and menu. The preceding code makes the following changes:

- Each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- The **Home** and **Privacy** menu entries are deleted.
- Entries are added for **About**, **Students**, **Courses**, **Instructors**, and **Departments**.

In `Pages/Index.cshtml`, replace the contents of the file with the following code:

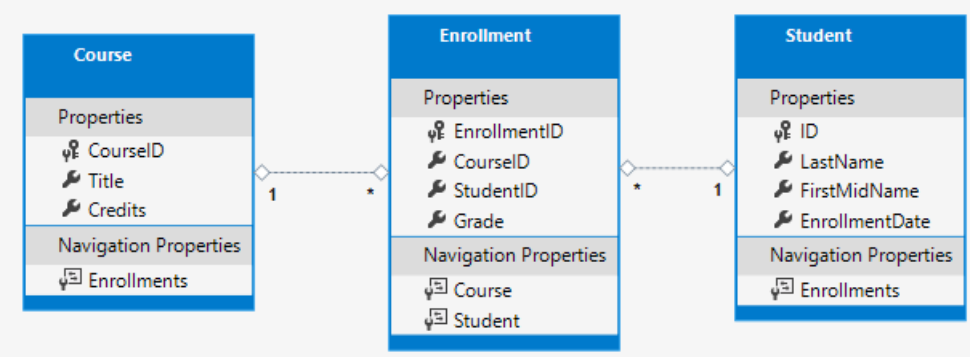
```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="row mb-auto">
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 mb-4 ">
                <p class="card-text">
                    Contoso University is a sample application that
                    demonstrates how to use Entity Framework Core in an
                    ASP.NET Core Razor Pages web app.
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column position-static">
                <p class="card-text mb-auto">
                    You can build the application by following the steps in a series of tutorials.
                </p>
                <p>
                    <a href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro" class="stretched-
link">See the tutorial</a>
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column">
                <p class="card-text mb-auto">
                    You can download the completed project from GitHub.
                </p>
                <p>
                    <a href="https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/data/ef-
rp/intro/samples" class="stretched-link">See project source code</a>
                </p>
            </div>
        </div>
    </div>
</div>
```

The preceding code replaces the text about ASP.NET Core with text about this app.

Run the app to verify that the home page appears.

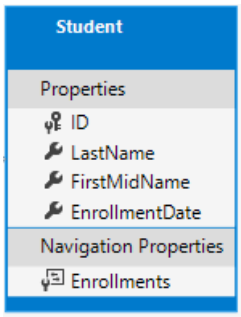
The data model

The following sections create a data model:



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

The Student entity



- Create a *Models* folder in the project folder.
- Create `Models/Student.cs` with the following code:

```
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; }
    }
}
```

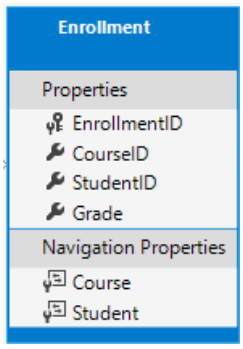
The `ID` property becomes the primary key column of the database table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key. So the alternative automatically recognized name for the `Student` class primary key is `StudentID`. For more information, see EF Core - Keys.

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 holds all of the `Enrollment` entities that are related to that Student. For example, if a Student row in the database has two related Enrollment rows, the `Enrollments` navigation property contains those two Enrollment entities.

In the database, an Enrollment row is related to a Student row if its `StudentID` column contains the student's ID value. For example, suppose a Student row has ID=1. Related Enrollment rows will have `StudentID` = 1. `StudentID` is a *foreign key* in the Enrollment table.

The `Enrollments` property is defined as `ICollection<Enrollment>` because there may be multiple related Enrollment entities. Other collection types can be used, such as `List<Enrollment>` or `HashSet<Enrollment>`. When `ICollection<Enrollment>` is used, EF Core creates a `HashSet<Enrollment>` collection by default.

The Enrollment entity



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

```
using System.ComponentModel.DataAnnotations;

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; }
    }
}
```

The `EnrollmentID` property is the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself. For a production data model, many developers choose one pattern and use it consistently. This tutorial uses both just to illustrate that both work. Using `ID` without `classname` makes it easier to implement some kinds of data model changes.

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.

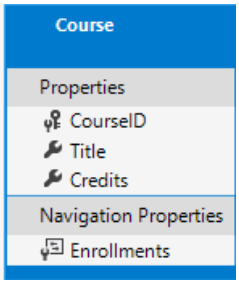
The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property contains a single `Student` entity.

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

EF Core interprets a property as a foreign key if it's named `<navigation property name><primary key property name>`. For example, `StudentID` is the foreign key for the `Student` navigation property, since the `Student` entity's primary key is `ID`. Foreign key properties can also be

named `<primary key property name>`. For example, `CourseID` since the `Course` entity's primary key is `CourseID`.

The Course entity



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

```
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.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the database generate it.

Build the project to validate that there are no compiler errors.

Scaffold Student pages

In this section, the ASP.NET Core scaffolding tool is used to generate:

- An EF Core `DbContext` class. The context is the main class that coordinates Entity Framework functionality for a given data model. It derives from the Microsoft.EntityFrameworkCore.DbContext class.
- Razor pages that handle Create, Read, Update, and Delete (CRUD) operations for the `Student` entity.

- Visual Studio
- Visual Studio Code

- Create a *Pages/Students* folder.
- In **Solution Explorer**, right-click the *Pages/Students* folder and select **Add** > **New Scaffolded Item**.
- In the **Add New Scaffold Item** dialog:
  - In the left tab, select **Installed** > **Common** > **Razor Pages**
  - Select **Razor Pages using Entity Framework (CRUD)** > **ADD**.
- In the **Add Razor Pages using Entity Framework (CRUD)** dialog:
  - In the **Model class** drop-down, select **Student (ContosoUniversity.Models)**.

- In the **Data context class** row, select the **+** (plus) sign.
  - Change the data context name to end in `SchoolContext` rather than `ContosoUniversityContext`.
    The updated context name: `ContosoUniversity.Data.SchoolContext`
  - Select **Add** to finish adding the data context class.
  - Select **Add** to finish the **Add Razor Pages** dialog.

If scaffolding fails with the error `'Install the package Microsoft.VisualStudio.Web.CodeGeneration.Design and try again.'`, run the scaffold tool again or see this GitHub issue.

The following packages are automatically installed:

- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.EntityFrameworkCore.Tools`
- `Microsoft.VisualStudio.Web.CodeGeneration.Design`

If the preceding step fails, build the project and retry the scaffold step.

The scaffolding process:

- Creates Razor pages in the *Pages/Students* folder:
  - `Create.cshtml` and `Create.cshtml.cs`
  - `Delete.cshtml` and `Delete.cshtml.cs`
  - `Details.cshtml` and `Details.cshtml.cs`
  - `Edit.cshtml` and `Edit.cshtml.cs`
  - `Index.cshtml` and `Index.cshtml.cs`
- Creates `Data/SchoolContext.cs`.
- Adds the context to dependency injection in `Startup.cs`.
- Adds a database connection string to `appsettings.json`.

Database connection string

The scaffolding tool generates a connection string in the `appsettings.json` file.

- Visual Studio
- Visual Studio Code

The connection string specifies SQL Server LocalDB:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=(localdb)\\mssqllocaldb;Database=CU-
1;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app

development, not production use. By default, LocalDB creates *.mdf* files in the `C:/Users/<user>` directory.

Update the database context class

The main class that coordinates EF Core functionality for a given data model is the database context class. The context is derived from Microsoft.EntityFrameworkCore.DbContext. The context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

Update `Data/SchoolContext.cs` with the following code:

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

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

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

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

The preceding code changes from the singular `DbSet<Student> Student` to the plural `DbSet<Student> Students`. To make the Razor Pages code match the new `DBSet` name, make a global change from: `_context.Student.` to: `_context.Students.`

There are 8 occurrences.

Because an entity set contains multiple entities, many developers prefer the `DBSet` property names should be plural.

The highlighted code:

- Creates a DbSet<TEntity> property for each entity set. In EF Core terminology:
  - An entity set typically corresponds to a database table.
  - An entity corresponds to a row in the table.
- Calls OnModelCreating. `OnModelCreating`:
  - Is called when `SchoolContext` has been initialized, but before the model has been locked down and used to initialize the context.
  - Is required because later in the tutorial the `Student` entity will have references to the other entities.

Build the project to verify there are no compiler errors.

Startup.cs

ASP.NET Core is built with dependency injection. Services such as the `SchoolContext` are registered with

dependency injection during app startup. Components that require these services, such as Razor Pages, are provided these services via constructor parameters. The constructor code that gets a database context instance is shown later in the tutorial.

The scaffolding tool automatically registered the context class with the dependency injection container.

- Visual Studio
- Visual Studio Code

The following highlighted lines were added by the scaffolder:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddDbContext<SchoolContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("SchoolContext")));
}
```

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

Add the database exception filter

Add AddDatabaseDeveloperPageExceptionFilter and UseMigrationsEndPoint as shown in the following code:

- Visual Studio
- Visual Studio Code

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddDbContext<SchoolContext>(options =>
       options.UseSqlServer(Configuration.GetConnectionString("SchoolContext")));

    services.AddDatabaseDeveloperPageExceptionFilter();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseMigrationsEndPoint();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

Add the Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore NuGet package.

In the Package Manager Console, enter the following to add the NuGet package:

```
Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
```

The `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore` NuGet package provides ASP.NET Core middleware for Entity Framework Core error pages. This middleware helps to detect and diagnose errors with Entity Framework Core migrations.

The `AddDatabaseDeveloperPageExceptionFilter` provides helpful error information in the development environment for EF migrations errors.

Create the database

Update `Program.cs` to create the database if it doesn't exist:

```
using ContosoUniversity.Data;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            CreateDbIfNotExists(host);

            host.Run();
        }

        private static void CreateDbIfNotExists(IHost host)
        {
            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;
                try
                {
                    var context = services.GetRequiredService<SchoolContext>();
                    context.Database.EnsureCreated();
                    // DbInitializer.Initialize(context);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred creating the DB.");
                }
            }
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

The EnsureCreated method takes no action if a database for the context exists. If no database exists, it creates the database and schema. `EnsureCreated` enables the following workflow for handling data model changes:

- Delete the database. Any existing data is lost.
- Change the data model. For example, add an `EmailAddress` field.
- Run the app.
- `EnsureCreated` creates a database with the new schema.

This workflow works early in development when the schema is rapidly evolving, as long as data doesn't need to be preserved. The situation is different when data that has been entered into the database needs to be preserved. When that is the case, use migrations.

Later in the tutorial series, the database is deleted that was created by `EnsureCreated` and migrations is used. A database that is created by `EnsureCreated` can't be updated by using migrations.

Test the app

- Run the app.
- Select the **Students** link and then **Create New**.
- Test the Edit, Details, and Delete links.

Seed the database

The `EnsureCreated` method creates an empty database. This section adds code that populates the database with test data.

Create `Data/DbInitializer.cs` with the following code:

```
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            // 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("2019-
09-01")},
                new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2017-
09-01")},
                new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2018-09-
01")},
                new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2017-
09-01")},
                new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2017-09-01")},
                new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2016-09-
01")},
                new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2018-09-
01")},
                new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2019-09-
01")}
            };

            context.Students.AddRange(students);
            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}
            };

            context.Courses.AddRange(courses);
            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},
            };

            context.Enrollments.AddRange(enrollments);
            context.SaveChanges();
        }
    }
}
```

The code checks if there are any students in the database. If there are no students, it adds test data to the database. It creates the test data in arrays rather than `List<T>` collections to optimize performance.

- In `Program.cs`, remove `//` from the `DbInitializer.Initialize` line:

```
context.Database.EnsureCreated();
DbInitializer.Initialize(context);
```

- Visual Studio
- Visual Studio Code

- Stop the app if it's running, and run the following command in the **Package Manager Console** (PMC):

```
Drop-Database -Confirm
```

- Respond with `Y` to delete the database.

- Restart the app.
- Select the Students page to see the seeded data.

View the database

- Visual Studio
- Visual Studio Code

- Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio.
- In SSOX, select **(localdb)\MSSQLLocalDB > Databases > SchoolContext-{GUID}**. The database name is generated from the context name provided earlier plus a dash and a GUID.
- Expand the **Tables** node.
- Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into the table.
- Right-click the **Student** table and click **View Code** to see how the `Student` model maps to the `Student` table schema.

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 doing 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 can handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. 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` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task OnGetAsync()
{
    Students = await _context.Students.ToListAsync();
}
```

- The `async` keyword tells the compiler to:
  - Generate callbacks for parts of the method body.
  - Create the Task object that's returned.
- The `Task` return type represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF Core context isn't thread safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the database.

For more information about asynchronous programming in .NET, see Async Overview and Asynchronous programming with async and await.

Performance considerations

In general, a web page shouldn't be loading an arbitrary number of rows. A query should use paging or a limiting approach. For example, the preceding query could use `Take` to limit the rows returned:

```
public async Task OnGetAsync()
{
    Student = await _context.Students.Take(10).ToListAsync();
}
```

Enumerating a large table in a view could return a partially constructed HTTP 200 response if a database exception occurs part way through the enumeration.

MaxModelBindingCollectionSize defaults to 1024. The following code sets `MaxModelBindingCollectionSize`:

```
public void ConfigureServices(IServiceCollection services)
{
    var myMaxModelBindingCollectionSize = Convert.ToInt32(
                Configuration["MyMaxModelBindingCollectionSize"] ?? "100");

    services.Configure<MvcOptions>(options =>
           options.MaxModelBindingCollectionSize = myMaxModelBindingCollectionSize);

    services.AddRazorPages();

    services.AddDbContext<SchoolContext>(options =>
          options.UseSqlServer(Configuration.GetConnectionString("SchoolContext")));

    services.AddDatabaseDeveloperPageExceptionFilter();
}
```

See Configuration for information on configuration settings like `MyMaxModelBindingCollectionSize`.

Paging is covered later in the tutorial.

For more information, see Performance considerations (EF).

SQL Logging of Entity Framework Core

Logging configuration is commonly provided by the `Logging` section of `appsettings.{Environment}.json` files. To log SQL statements, add `"Microsoft.EntityFrameworkCore.Database.Command": "Information"` to the `appsettings.Development.json` file:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=MyDB-
2;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
     ,"Microsoft.EntityFrameworkCore.Database.Command": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

With the preceding JSON, SQL statements are displayed on the command line and in the Visual Studio output window.

For more information, see Logging in .NET Core and ASP.NET Core and this GitHub issue.

Next steps

Use SQLite for development, SQL Server for production

NEXT TUTORIAL

This is the first in a series of tutorials that show how to use Entity Framework (EF) Core in an ASP.NET Core Razor Pages app. The tutorials build a web site for a fictional Contoso University. The site includes functionality such as student admission, course creation, and instructor assignments. The tutorial uses the code first approach. For information on following this tutorial using the database first approach, see this Github issue.

Download or view the completed app. Download instructions.

Prerequisites

- If you're new to Razor Pages, go through the Get started with Razor Pages tutorial series before starting this one.

- Visual Studio
- Visual Studio Code

- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core 3.0 SDK

Database engines

The Visual Studio instructions use SQL Server LocalDB, a version of SQL Server Express that runs only on Windows.

The Visual Studio Code instructions use SQLite, a cross-platform database engine.

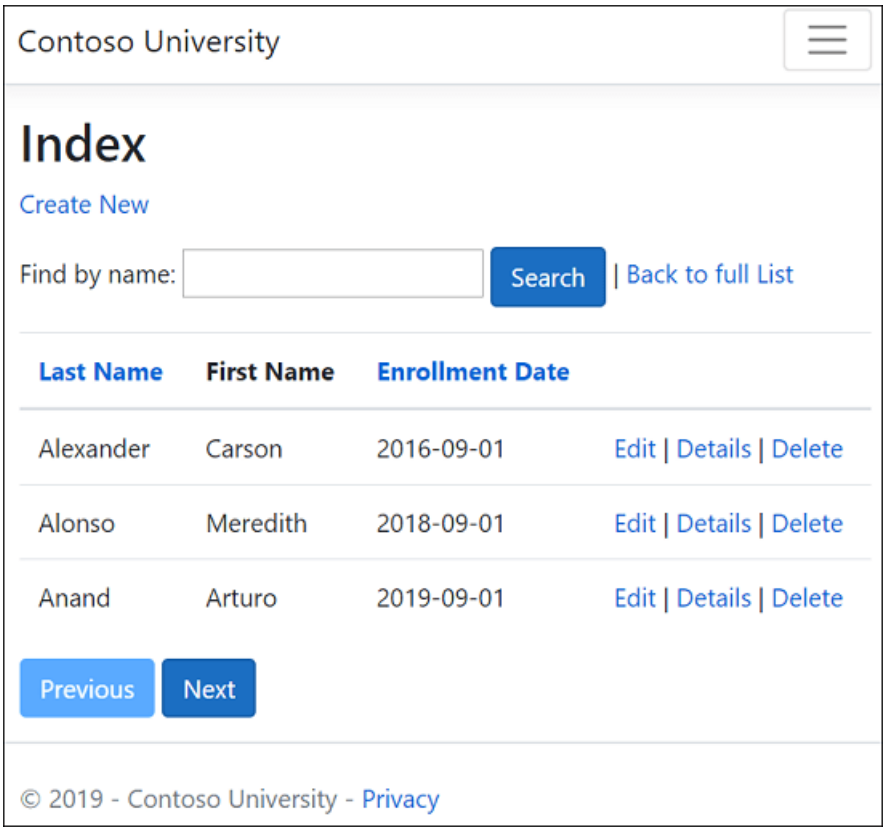
If you choose to use SQLite, download and install a third-party tool for managing and viewing a SQLite database, such as DB Browser for SQLite.

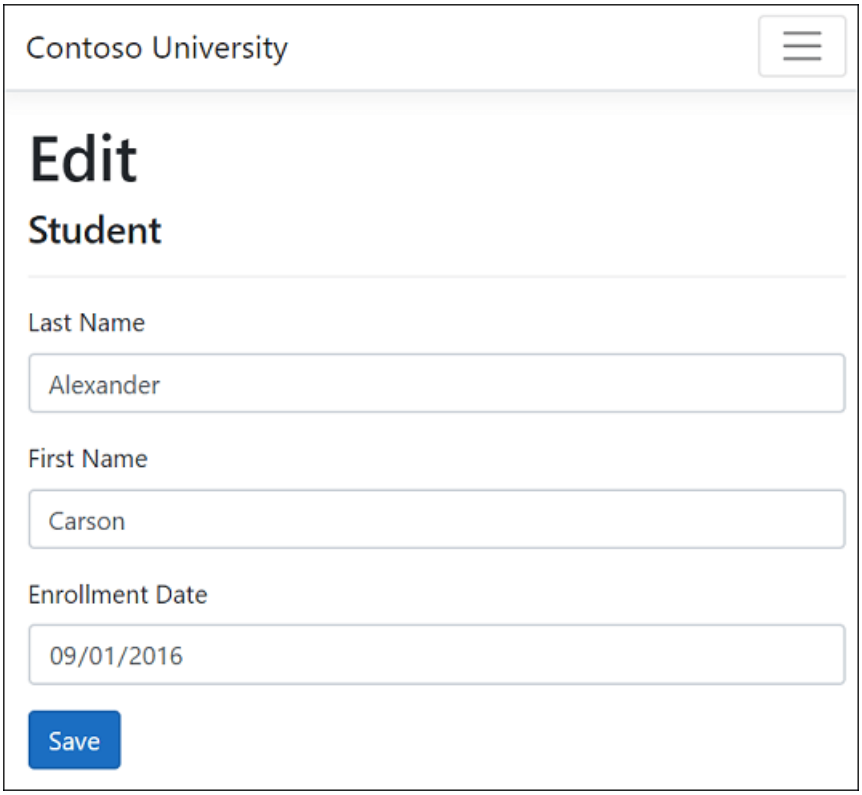
Troubleshooting

If you run into a problem you can't resolve, compare your code to the completed project. A good way to get help is by posting a question to StackOverflow.com, using the ASP.NET Core tag or the EF Core tag.

The sample app

The app built in these tutorials is a basic university web site. Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.





The UI style of this site is based on the built-in project templates. The tutorial's focus is on how to use EF Core, not how to customize the UI.

Follow the link at the top of the page to get the source code for the completed project. The *cu30* folder has the code for the ASP.NET Core 3.0 version of the tutorial. Files that reflect the state of the code for tutorials 1-7 can be found in the *cu30snapshots* folder.

- Visual Studio
- Visual Studio Code

To run the app after downloading the completed project:

- Build the project.
- In Package Manager Console (PMC) run the following command:

```
Update-Database
```

- Run the project to seed the database.

Create the web app project

- Visual Studio
- Visual Studio Code

- From the Visual Studio **File** menu, select **New** > **Project**.
- Select **ASP.NET Core Web Application**.
- Name the project *ContosoUniversity*. It's important to use this exact name including capitalization, so the namespaces match when code is copied and pasted.
- Select **.NET Core** and **ASP.NET Core 3.0** in the dropdowns, and then select **Web Application**.

Set up the site style

Set up the site header, footer, and menu by updating `Pages/Shared/_Layout.cshtml`:

- Change each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- Delete the **Home** and **Privacy** menu entries, and add entries for **About**, **Students**, **Courses**, **Instructors**, and **Departments**.

The changes are highlighted.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-
shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-page="/Index">Contoso University</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-
collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Students/Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/Courses/Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Instructors/Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Departments/Index">Departments</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2019 - Contoso University - <a asp-area="" asp-page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

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

In `Pages/Index.cshtml`, replace the contents of the file with the following code to replace the text about ASP.NET

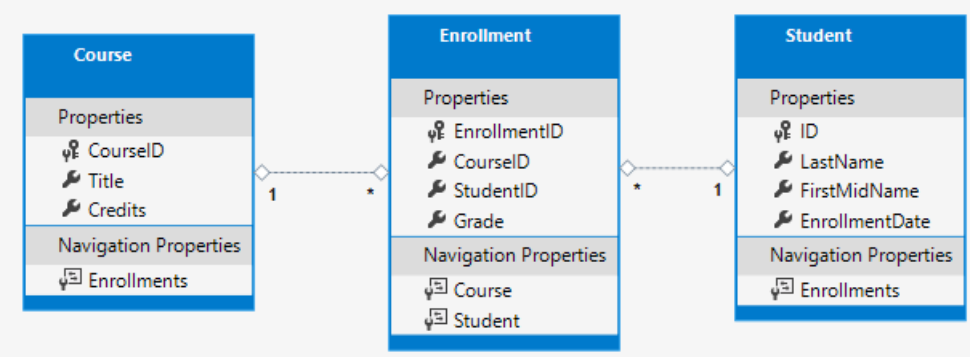
Core with text about this app:

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="row mb-auto">
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 mb-4 ">
                <p class="card-text">
                    Contoso University is a sample application that
                    demonstrates how to use Entity Framework Core in an
                    ASP.NET Core Razor Pages web app.
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column position-static">
                <p class="card-text mb-auto">
                    You can build the application by following the steps in a series of tutorials.
                </p>
                <p>
                    <a href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro" class="stretched-
link">See the tutorial</a>
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column">
                <p class="card-text mb-auto">
                    You can download the completed project from GitHub.
                </p>
                <p>
                    <a href="https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/data/ef-
rp/intro/samples" class="stretched-link">See project source code</a>
                </p>
            </div>
        </div>
    </div>
</div>
```

Run the app to verify that the home page appears.

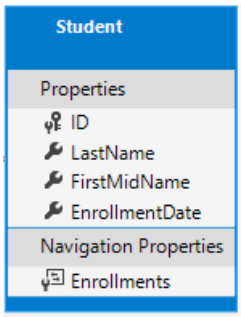
The data model

The following sections create a data model:



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

The Student entity



- Create a *Models* folder in the project folder.
- Create `Models/Student.cs` with the following code:

```
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; }
    }
}
```

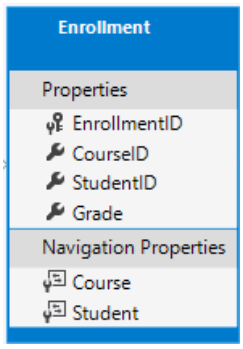
The `ID` property becomes the primary key column of the database table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key. So the alternative automatically recognized name for the `Student` class primary key is `StudentID`. For more information, see EF Core - Keys.

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 holds all of the `Enrollment` entities that are related to that Student. For example, if a Student row in the database has two related Enrollment rows, the `Enrollments` navigation property contains those two Enrollment entities.

In the database, an Enrollment row is related to a Student row if its StudentID column contains the student's ID value. For example, suppose a Student row has ID=1. Related Enrollment rows will have StudentID = 1. StudentID is a *foreign key* in the Enrollment table.

The `Enrollments` property is defined as `ICollection<Enrollment>` because there may be multiple related Enrollment entities. You can use other collection types, such as `List<Enrollment>` or `HashSet<Enrollment>`. When `ICollection<Enrollment>` is used, EF Core creates a `HashSet<Enrollment>` collection by default.

The Enrollment entity



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

```
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 is the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself. For a production data model, choose one pattern and use it consistently. This tutorial uses both just to illustrate that both work. Using `ID` without `classname` makes it easier to implement some kinds of data model changes.

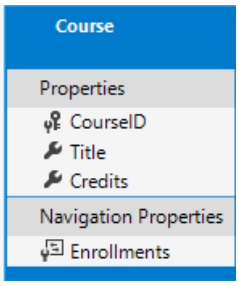
The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade—null means a grade isn't known or hasn't been assigned yet.

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 contains a single `Student` entity.

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

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

The Course entity



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

```
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.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the database generate it.

Build the project to validate that there are no compiler errors.

Scaffold Student pages

In this section, you use the ASP.NET Core scaffolding tool to generate:

- An EF Core *context* class. The context is the main class that coordinates Entity Framework functionality for a given data model. It derives from the `Microsoft.EntityFrameworkCore.DbContext` class.
- Razor pages that handle Create, Read, Update, and Delete (CRUD) operations for the `Student` entity.

- Visual Studio
- Visual Studio Code

- Create a *Students* folder in the *Pages* folder.
- In **Solution Explorer**, right-click the *Pages/Students* folder and select **Add** > **New Scaffolded Item**.
- In the **Add Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD)** > **ADD**.
- In the **Add Razor Pages using Entity Framework (CRUD)** dialog:
  - In the **Model class** drop-down, select **Student (ContosoUniversity.Models)**.
  - In the **Data context class** row, select the **+** (plus) sign.
  - Change the data context name from *ContosoUniversity.Models.ContosoUniversityContext* to *ContosoUniversity.Data.SchoolContext*.
  - Select **Add**.

The following packages are automatically installed:

- `Microsoft.VisualStudio.Web.CodeGeneration.Design`
- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.Extensions.Logging.Debug`
- `Microsoft.EntityFrameworkCore.Tools`

If you have a problem with the preceding step, build the project and retry the scaffold step.

The scaffolding process:

- Creates Razor pages in the *Pages/Students* folder:
  - `Create.cshtml` and `Create.cshtml.cs`
  - `Delete.cshtml` and `Delete.cshtml.cs`
  - `Details.cshtml` and `Details.cshtml.cs`
  - `Edit.cshtml` and `Edit.cshtml.cs`
  - `Index.cshtml` and `Index.cshtml.cs`
- Creates `Data/SchoolContext.cs`.
- Adds the context to dependency injection in `Startup.cs`.
- Adds a database connection string to `appsettings.json`.

Database connection string

- Visual Studio
- Visual Studio Code

The `appsettings.json` file specifies the connection string SQL Server LocalDB.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=
(localdb)\\mssqllocaldb;Database=SchoolContext6;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. By default, LocalDB creates *.mdf* files in the `C:/Users/<user>` directory.

Update the database context class

The main class that coordinates EF Core functionality for a given data model is the database context class. The context is derived from Microsoft.EntityFrameworkCore.DbContext. The context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

Update `Data/SchoolContext.cs` with the following code:

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

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

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

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

The highlighted code creates a DbSet<TEntity> property for each entity set. In EF Core terminology:

- An entity set typically corresponds to a database table.
- An entity corresponds to a row in the table.

Since an entity set contains multiple entities, the DBSet properties should be plural names. Since the scaffolding tool created a `Student` DBSet, this step changes it to plural `Students`.

To make the Razor Pages code match the new DBSet name, make a global change across the whole project of `_context.Student` to `_context.Students`. There are 8 occurrences.

Build the project to verify there are no compiler errors.

Startup.cs

ASP.NET Core is built with dependency injection. Services (such as the EF Core database context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a database context instance is shown later in the tutorial.

The scaffolding tool automatically registered the context class with the dependency injection container.

- Visual Studio
- Visual Studio Code

- In `ConfigureServices`, the highlighted lines were added by the scaffolder:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddDbContext<SchoolContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("SchoolContext")));
}
```

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

Create the database

Update `Program.cs` to create the database if it doesn't exist:

```
using ContosoUniversity.Data;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            CreateDbIfNotExists(host);

            host.Run();
        }

        private static void CreateDbIfNotExists(IHost host)
        {
            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;
                try
                {
                    var context = services.GetRequiredService<SchoolContext>();
                    context.Database.EnsureCreated();
                    // DbInitializer.Initialize(context);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred creating the DB.");
                }
            }
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

The EnsureCreated method takes no action if a database for the context exists. If no database exists, it creates the database and schema. `EnsureCreated` enables the following workflow for handling data model changes:

- Delete the database. Any existing data is lost.
- Change the data model. For example, add an `EmailAddress` field.
- Run the app.
- `EnsureCreated` creates a database with the new schema.

This workflow works well early in development when the schema is rapidly evolving, as long as you don't need to preserve data. The situation is different when data that has been entered into the database needs to be preserved. When that is the case, use migrations.

Later in the tutorial series, you delete the database that was created by `EnsureCreated` and use migrations instead. A database that is created by `EnsureCreated` can't be updated by using migrations.

Test the app

- Run the app.
- Select the **Students** link and then **Create New**.
- Test the Edit, Details, and Delete links.

Seed the database

The `EnsureCreated` method creates an empty database. This section adds code that populates the database with test data.

Create `Data/DbInitializer.cs` with the following code:

```
using ContosoUniversity.Data;
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("2019-
09-01")},
                new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2017-
09-01")},
                new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2018-09-
01")},
                new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2017-
09-01")},
                new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2017-09-01")},
                new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2016-09-
01")},
                new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2018-09-
01")},
                new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2019-09-
01")}
            };

            context.Students.AddRange(students);
            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=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}
            };

            context.Courses.AddRange(courses);
            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},
            };

            context.Enrollments.AddRange(enrollments);
            context.SaveChanges();
        }
    }
}
```

The code checks if there are any students in the database. If there are no students, it adds test data to the database. It creates the test data in arrays rather than `List<T>` collections to optimize performance.

- In `Program.cs`, replace the `EnsureCreated` call with a `DbInitializer.Initialize` call:

```
// context.Database.EnsureCreated();
DbInitializer.Initialize(context);
```

- Visual Studio
- Visual Studio Code

Stop the app if it's running, and run the following command in the **Package Manager Console** (PMC):

```
Drop-Database
```

- Restart the app.
- Select the Students page to see the seeded data.

View the database

- Visual Studio
- Visual Studio Code

- Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio.
- In SSOX, select **(localdb)\MSSQLLocalDB > Databases > SchoolContext-{GUID}**. The database name is generated from the context name you provided earlier plus a dash and a GUID.

- Expand the **Tables** node.
- Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into the table.
- Right-click the **Student** table and click **View Code** to see how the `Student` model maps to the `Student` table schema.

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 can handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. 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.

```
public async Task OnGetAsync()
{
    Students = await _context.Students.ToListAsync();
}
```

- The `async` keyword tells the compiler to:
  - Generate callbacks for parts of the method body.
  - Create the Task object that's returned.
- The `Task<T>` return type represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF Core context isn't thread safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the database.

For more information about asynchronous programming in .NET, see Async Overview and Asynchronous programming with async and await.

Next steps

NEXT

TUTORIAL

Part 2, Razor Pages with EF Core in ASP.NET Core - CRUD

By Tom Dykstra, Jeremy Likness, and Jon P Smith

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

In this tutorial, the scaffolded CRUD (create, read, update, delete) code is reviewed and customized.

No repository

Some developers use a service layer or repository pattern to create an abstraction layer between the UI (Razor Pages) and the data access layer. This tutorial doesn't do that. To minimize complexity and keep the tutorial focused on EF Core, EF Core code is added directly to the page model classes.

Update the Details page

The scaffolded code for the Students pages doesn't include enrollment data. In this section, enrollments are added to the `Details` page.

Read enrollments

To display a student's enrollment data on the page, the enrollment data must be read. The scaffolded code in `Pages/Students/Details.cshtml.cs` reads only the `Student` data, without the `Enrollment` data:

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

    Student = await _context.Students.FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

Replace the `OnGetAsync` method with the following code to read enrollment data for the selected student. The changes are highlighted.

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

    Student = await _context.Students
        .Include(s => s.Enrollments)
        .ThenInclude(e => e.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

The Include and ThenInclude methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. These methods are examined in detail in the Read related data tutorial.

The AsNoTracking method improves performance in scenarios where the entities returned are not updated in the current context. `AsNoTracking` is discussed later in this tutorial.

Display enrollments

Replace the code in `Pages/Students/Details.cshtml` with the following code to display a list of enrollments. The changes are highlighted.

```
@page
@model ContosoUniversity.Pages.Students.DetailsModel

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

<h1>Details</h1>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.Enrollments)
        </dt>
        <dd class="col-sm-10">
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Student.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Student.ID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

The preceding 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 app, select the **Students** tab, and click the **Details** link for a student. The list of courses and grades for

the selected student is displayed.

Ways to read one entity

The generated code uses FirstOrDefaultAsync to read one entity. This method returns null if nothing is found; otherwise, it returns the first row found that satisfies the query filter criteria. `FirstOrDefaultAsync` is generally a better choice than the following alternatives:

- SingleOrDefaultAsync - Throws an exception if there's more than one entity that satisfies the query filter. To determine if more than one row could be returned by the query, `SingleOrDefaultAsync` tries to fetch multiple rows. This extra work is unnecessary if the query can only return one entity, as when it searches on a unique key.
- FindAsync - Finds an entity with the primary key (PK). If an entity with the PK is being tracked by the context, it's returned without a request to the database. This method is optimized to look up a single entity, but you can't call `Include` with `FindAsync`. So if related data is needed, `FirstOrDefaultAsync` is the better choice.

Route data vs. query string

The URL for the Details page is `https://localhost:<port>/Students/Details?id=1`. The entity's primary key value is in the query string. Some developers prefer to pass the key value in route data: `https://localhost:<port>/Students/Details/1`. For more information, see Update the generated code.

Update the Create page

The scaffolded `OnPostAsync` code for the Create page is vulnerable to overposting. Replace the `OnPostAsync` method in `Pages/Students/Create.cshtml.cs` with the following code.

```
public async Task<IActionResult> OnPostAsync()
{
    var emptyStudent = new Student();

    if (await TryUpdateModelAsync<Student>(
        emptyStudent,
        "student",   // Prefix for form value.
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        _context.Students.Add(emptyStudent);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

TryUpdateModelAsync

The preceding code creates a Student object and then uses posted form fields to update the Student object's properties. The TryUpdateModelAsync method:

- Uses the posted form values from the PageContext property in the PageModel.
- Updates only the properties listed (`s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate`).
- Looks for form fields with a "student" prefix. For example, `Student.FirstMidName`. It's not case sensitive.
- Uses the model binding system to convert form values from strings to the types in the `Student` model. For example, `EnrollmentDate` is converted to `DateTime`.

Run the app, and create a student entity to test the Create page.

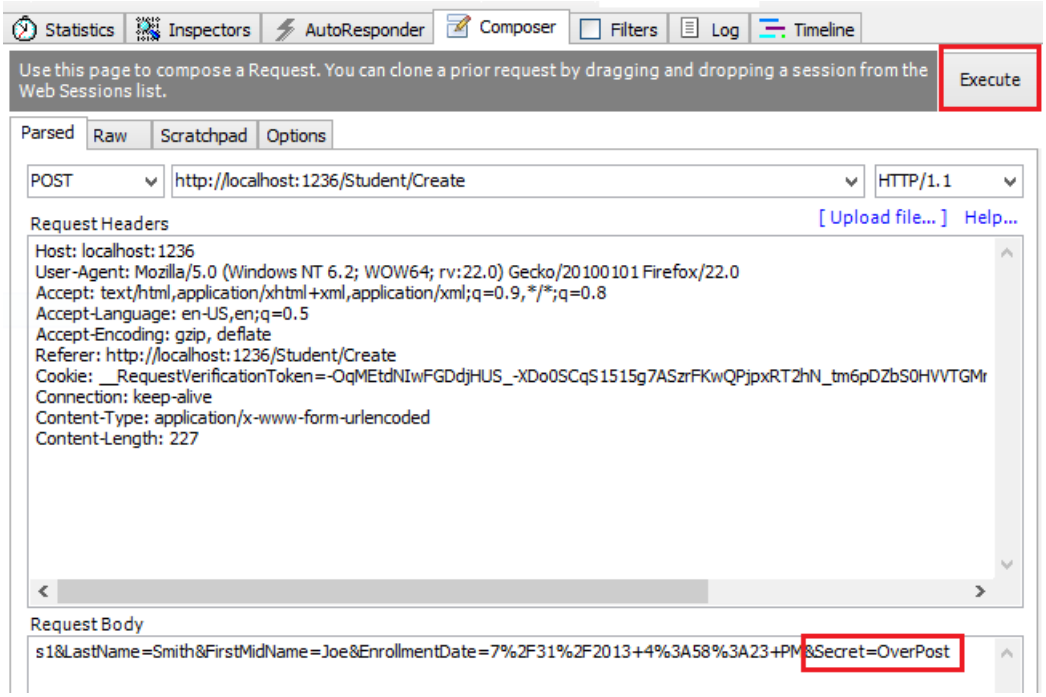
Overposting

Using `TryUpdateModel` to update fields with posted values is a security best practice because it prevents overposting. For example, suppose the Student entity includes a `Secret` property that this web page shouldn't update or add:

```
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 the app doesn't have a `Secret` field on the create or update Razor Page, a hacker could set the `Secret` value by overposting. A hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. The original code doesn't limit the fields that the model binder uses when it creates a Student instance.

Whatever value the hacker specified for the `Secret` form field is updated in the database. The following image shows the Fiddler tool adding the `Secret` field, with the value "OverPost", to the posted form values.



The value "OverPost" is successfully added to the `Secret` property of the inserted row. That happens even though the app designer never intended the `Secret` property to be set with the Create page.

View model

View models provide an alternative way to prevent overposting.

The application model is often called the domain model. The domain model typically contains all the properties required by the corresponding entity in the database. The view model contains only the properties needed for the UI page, for example, the Create page.

In addition to the view model, some apps use a binding model or input model to pass data between the Razor Pages page model class and the browser.

Consider the following `StudentVM` view model:

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

The following code uses the `StudentVM` view model to create a new student:

```
[BindProperty]
public StudentVM StudentVM { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var entry = _context.Add(new Student());
    entry.CurrentValues.SetValues(StudentVM);
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
```

The SetValues method sets the values of this object by reading values from another PropertyValues object. `SetValues` uses property name matching. The view model type:

- Doesn't need to be related to the model type.
- Needs to have properties that match.

Using `StudentVM` requires the Create page use `StudentVM` rather than `Student`:

```
@page
@model CreateVMModel

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

<h1>Create</h1>

<h4>Student</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="StudentVM.LastName" class="control-label"></label>
                <input asp-for="StudentVM.LastName" class="form-control" />
                <span asp-validation-for="StudentVM.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StudentVM.FirstMidName" class="control-label"></label>
                <input asp-for="StudentVM.FirstMidName" class="form-control" />
                <span asp-validation-for="StudentVM.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StudentVM.EnrollmentDate" class="control-label"></label>
                <input asp-for="StudentVM.EnrollmentDate" class="form-control" />
                <span asp-validation-for="StudentVM.EnrollmentDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Update the Edit page

In `Pages/Students/Edit.cshtml.cs`, replace the `OnGetAsync` and `OnPostAsync` methods with the following code.

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

    Student = await _context.Students.FindAsync(id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}

public async Task<IActionResult> OnPostAsync(int id)
{
    var studentToUpdate = await _context.Students.FindAsync(id);

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

    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "student",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

The code changes are similar to the Create page with a few exceptions:

- `FirstOrDefaultAsync` has been replaced with FindAsync. When you don't have to include related data, `FindAsync` is more efficient.
- `OnPostAsync` has an `id` parameter.
- The current student is fetched from the database, rather than creating an empty student.

Run the app, and test it by creating and editing a student.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database. This tracking information determines what happens when SaveChangesAsync is called. For example, when a new entity is passed to the AddAsync method, that entity's state is set to Added. When `SaveChangesAsync` is called, the database context issues a SQL `INSERT` command.

An entity may be in one of the following states:

- `Added`: The entity doesn't yet exist in the database. The `SaveChanges` method issues an `INSERT` statement.
- `Unchanged`: No changes need to be saved with this entity. An entity has this status when it's read from the database.
- `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 app, state changes are typically set automatically. An entity is read, changes are made, and the entity state is automatically changed to `Modified`. Calling `SaveChanges` generates a SQL `UPDATE` statement that updates only the changed properties.

In a web app, the `DbContext` that reads an entity and displays the data is disposed after a page is rendered. When a page's `OnPostAsync` method is called, a new web request is made and with a new instance of the `DbContext`. Rereading the entity in that new context simulates desktop processing.

Update the Delete page

In this section, a custom error message is implemented when the call to `SaveChanges` fails.

Replace the code in `Pages/Students/Delete.cshtml.cs` with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using System;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;
        private readonly ILogger<DeleteModel> _logger;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context,
                           ILogger<DeleteModel> logger)
        {
            _context = context;
            _logger = logger;
        }

        [BindProperty]
        public Student Student { get; set; }
        public string ErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id, bool? saveChangesError = false)
        {
            if (id == null)
            {
                return NotFound();
            }

            Student = await _context.Students
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.ID == id);

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

            if (saveChangesError.GetValueOrDefault())
            {
                ErrorMessage = String.Format("Delete {ID} failed. Try again", id);
```

```
            }

            return Page();
        }

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

            var student = await _context.Students.FindAsync(id);

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

            try
            {
                _context.Students.Remove(student);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            catch (DbUpdateException ex)
            {
                _logger.LogError(ex, ErrorMessage);

                return RedirectToAction("./Delete",
                                        new { id, saveChangesError = true });
            }
        }
    }
}
```

The preceding code:

- Adds Logging.
- Adds the optional parameter `saveChangesError` to the `OnGetAsync` method signature. `saveChangesError` indicates whether the method was called after a failure to delete the student object.

The delete operation might fail because of transient network problems. Transient network errors are more likely when the database is in the cloud. The `saveChangesError` parameter is `false` when the Delete page `OnGetAsync` is called from the UI. When `OnGetAsync` is called by `OnPostAsync` because the delete operation failed, the `saveChangesError` parameter is `true`.

The `OnPostAsync` method 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. If `Remove` fails:

- The database exception is caught.
- The Delete pages `OnGetAsync` method is called with `saveChangesError=true`.

Add an error message to `Pages/Students/Delete.cshtml`:

```
@page
@model ContosoUniversity.Pages.Students.DeleteModel

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

<h1>Delete</h1>

<p class="text-danger">@Model.ErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Student.ID" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Run the app and delete a student to test the Delete page.

Next steps

PREVIOUS TUTORIAL | NEXT TUTORIAL

In this tutorial, the scaffolded CRUD (create, read, update, delete) code is reviewed and customized.

No repository

Some developers use a service layer or repository pattern to create an abstraction layer between the UI (Razor Pages) and the data access layer. This tutorial doesn't do that. To minimize complexity and keep the tutorial focused on EF Core, EF Core code is added directly to the page model classes.

Update the Details page

The scaffolded code for the Students pages doesn't include enrollment data. In this section, enrollments are added to the `Details` page.

Read enrollments

To display a student's enrollment data on the page, the enrollment data must be read. The scaffolded code in `Pages/Students/Details.cshtml.cs` reads only the `Student` data, without the `Enrollment` data:

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

    Student = await _context.Students.FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

Replace the `OnGetAsync` method with the following code to read enrollment data for the selected student. The changes are highlighted.

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

    Student = await _context.Students
        .Include(s => s.Enrollments)
        .ThenInclude(e => e.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

The Include and ThenInclude methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. These methods are examined in detail in the Read related data tutorial.

The AsNoTracking method improves performance in scenarios where the entities returned are not updated in the current context. `AsNoTracking` is discussed later in this tutorial.

Display enrollments

Replace the code in `Pages/Students/Details.cshtml` with the following code to display a list of enrollments. The changes are highlighted.

```
@page
@model ContosoUniversity.Pages.Students.DetailsModel

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

<h1>Details</h1>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.Enrollments)
        </dt>
        <dd class="col-sm-10">
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Student.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Student.ID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

The preceding 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 app, select the **Students** tab, and click the **Details** link for a student. The list of courses and grades for

the selected student is displayed.

Ways to read one entity

The generated code uses FirstOrDefaultAsync to read one entity. This method returns null if nothing is found; otherwise, it returns the first row found that satisfies the query filter criteria. `FirstOrDefaultAsync` is generally a better choice than the following alternatives:

- SingleOrDefaultAsync - Throws an exception if there's more than one entity that satisfies the query filter. To determine if more than one row could be returned by the query, `SingleOrDefaultAsync` tries to fetch multiple rows. This extra work is unnecessary if the query can only return one entity, as when it searches on a unique key.
- FindAsync - Finds an entity with the primary key (PK). If an entity with the PK is being tracked by the context, it's returned without a request to the database. This method is optimized to look up a single entity, but you can't call `Include` with `FindAsync`. So if related data is needed, `FirstOrDefaultAsync` is the better choice.

Route data vs. query string

The URL for the Details page is `https://localhost:<port>/Students/Details?id=1`. The entity's primary key value is in the query string. Some developers prefer to pass the key value in route data: `https://localhost:<port>/Students/Details/1`. For more information, see Update the generated code.

Update the Create page

The scaffolded `OnPostAsync` code for the Create page is vulnerable to overposting. Replace the `OnPostAsync` method in `Pages/Students/Create.cshtml.cs` with the following code.

```
public async Task<IActionResult> OnPostAsync()
{
    var emptyStudent = new Student();

    if (await TryUpdateModelAsync<Student>(
        emptyStudent,
        "student",   // Prefix for form value.
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        _context.Students.Add(emptyStudent);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

TryUpdateModelAsync

The preceding code creates a Student object and then uses posted form fields to update the Student object's properties. The TryUpdateModelAsync method:

- Uses the posted form values from the PageContext property in the PageModel.
- Updates only the properties listed (`s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate`).
- Looks for form fields with a "student" prefix. For example, `Student.FirstMidName`. It's not case sensitive.
- Uses the model binding system to convert form values from strings to the types in the `Student` model. For example, `EnrollmentDate` is converted to `DateTime`.

Run the app, and create a student entity to test the Create page.

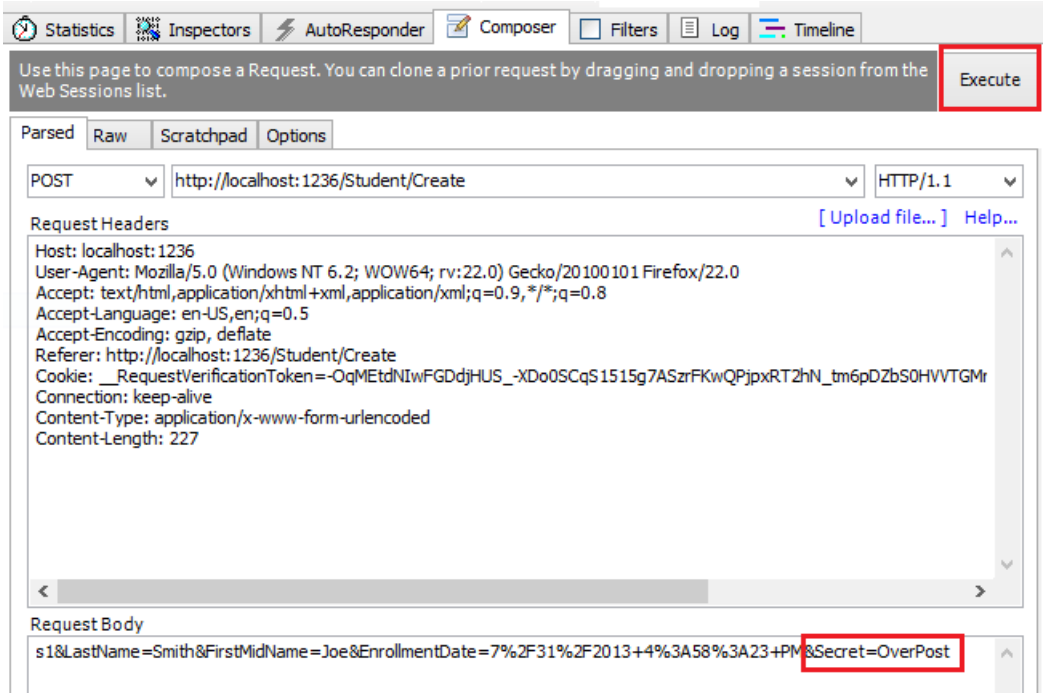
Overposting

Using `TryUpdateModel` to update fields with posted values is a security best practice because it prevents overposting. For example, suppose the Student entity includes a `Secret` property that this web page shouldn't update or add:

```
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 the app doesn't have a `Secret` field on the create or update Razor Page, a hacker could set the `Secret` value by overposting. A hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. The original code doesn't limit the fields that the model binder uses when it creates a Student instance.

Whatever value the hacker specified for the `Secret` form field is updated in the database. The following image shows the Fiddler tool adding the `Secret` field, with the value "OverPost", to the posted form values.



The value "OverPost" is successfully added to the `Secret` property of the inserted row. That happens even though the app designer never intended the `Secret` property to be set with the Create page.

View model

View models provide an alternative way to prevent overposting.

The application model is often called the domain model. The domain model typically contains all the properties required by the corresponding entity in the database. The view model contains only the properties needed for the UI page, for example, the Create page.

In addition to the view model, some apps use a binding model or input model to pass data between the Razor Pages page model class and the browser.

Consider the following `StudentVM` view model:

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

The following code uses the `StudentVM` view model to create a new student:

```
[BindProperty]
public StudentVM StudentVM { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var entry = _context.Add(new Student());
    entry.CurrentValues.SetValues(StudentVM);
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
```

The SetValues method sets the values of this object by reading values from another PropertyValues object. `SetValues` uses property name matching. The view model type:

- Doesn't need to be related to the model type.
- Needs to have properties that match.

Using `StudentVM` requires the Create page use `StudentVM` rather than `Student`:

```
@page
@model CreateVMModel

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

<h1>Create</h1>

<h4>Student</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="StudentVM.LastName" class="control-label"></label>
                <input asp-for="StudentVM.LastName" class="form-control" />
                <span asp-validation-for="StudentVM.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StudentVM.FirstMidName" class="control-label"></label>
                <input asp-for="StudentVM.FirstMidName" class="form-control" />
                <span asp-validation-for="StudentVM.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StudentVM.EnrollmentDate" class="control-label"></label>
                <input asp-for="StudentVM.EnrollmentDate" class="form-control" />
                <span asp-validation-for="StudentVM.EnrollmentDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Update the Edit page

In `Pages/Students/Edit.cshtml.cs`, replace the `OnGetAsync` and `OnPostAsync` methods with the following code.

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

    Student = await _context.Students.FindAsync(id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}

public async Task<IActionResult> OnPostAsync(int id)
{
    var studentToUpdate = await _context.Students.FindAsync(id);

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

    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "student",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

The code changes are similar to the Create page with a few exceptions:

- `FirstOrDefaultAsync` has been replaced with FindAsync. When you don't have to include related data, `FindAsync` is more efficient.
- `OnPostAsync` has an `id` parameter.
- The current student is fetched from the database, rather than creating an empty student.

Run the app, and test it by creating and editing a student.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database. This tracking information determines what happens when SaveChangesAsync is called. For example, when a new entity is passed to the AddAsync method, that entity's state is set to Added. When `SaveChangesAsync` is called, the database context issues a SQL `INSERT` command.

An entity may be in one of the following states:

- `Added`: The entity doesn't yet exist in the database. The `SaveChanges` method issues an `INSERT` statement.
- `Unchanged`: No changes need to be saved with this entity. An entity has this status when it's read from the database.
- `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 app, state changes are typically set automatically. An entity is read, changes are made, and the entity state is automatically changed to `Modified`. Calling `SaveChanges` generates a SQL `UPDATE` statement that updates only the changed properties.

In a web app, the `DbContext` that reads an entity and displays the data is disposed after a page is rendered. When a page's `OnPostAsync` method is called, a new web request is made and with a new instance of the `DbContext`. Rereading the entity in that new context simulates desktop processing.

Update the Delete page

In this section, a custom error message is implemented when the call to `SaveChanges` fails.

Replace the code in `Pages/Students/Delete.cshtml.cs` with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using System;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;
        private readonly ILogger<DeleteModel> _logger;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context,
                           ILogger<DeleteModel> logger)
        {
            _context = context;
            _logger = logger;
        }

        [BindProperty]
        public Student Student { get; set; }
        public string ErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id, bool? saveChangesError = false)
        {
            if (id == null)
            {
                return NotFound();
            }

            Student = await _context.Students
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.ID == id);

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

            if (saveChangesError.GetValueOrDefault())
            {
                ErrorMessage = String.Format("Delete {ID} failed. Try again", id);
```

```
            }

            return Page();
        }

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

            var student = await _context.Students.FindAsync(id);

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

            try
            {
                _context.Students.Remove(student);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            catch (DbUpdateException ex)
            {
                _logger.LogError(ex, ErrorMessage);

                return RedirectToAction("./Delete",
                                        new { id, saveChangesError = true });
            }
        }
    }
}
```

The preceding code:

- Adds Logging.
- Adds the optional parameter `saveChangesError` to the `OnGetAsync` method signature. `saveChangesError` indicates whether the method was called after a failure to delete the student object.

The delete operation might fail because of transient network problems. Transient network errors are more likely when the database is in the cloud. The `saveChangesError` parameter is `false` when the Delete page `OnGetAsync` is called from the UI. When `OnGetAsync` is called by `OnPostAsync` because the delete operation failed, the `saveChangesError` parameter is `true`.

The `OnPostAsync` method 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. If `Remove` fails:

- The database exception is caught.
- The Delete pages `OnGetAsync` method is called with `saveChangesError=true`.

Add an error message to `Pages/Students/Delete.cshtml`:

```
@page
@model ContosoUniversity.Pages.Students.DeleteModel

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

<h1>Delete</h1>

<p class="text-danger">@Model.ErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Student.ID" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Run the app and delete a student to test the Delete page.

Next steps

PREVIOUS TUTORIAL | NEXT TUTORIAL

In this tutorial, the scaffolded CRUD (create, read, update, delete) code is reviewed and customized.

No repository

Some developers use a service layer or repository pattern to create an abstraction layer between the UI (Razor Pages) and the data access layer. This tutorial doesn't do that. To minimize complexity and keep the tutorial focused on EF Core, EF Core code is added directly to the page model classes.

Update the Details page

The scaffolded code for the Students pages doesn't include enrollment data. In this section, enrollments are added to the Details page.

Read enrollments

To display a student's enrollment data on the page, the enrollment data needs to be read. The scaffolded code in `Pages/Students/Details.cshtml.cs` reads only the Student data, without the Enrollment data:

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

    Student = await _context.Students.FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

Replace the `OnGetAsync` method with the following code to read enrollment data for the selected student. The changes are highlighted.

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

    Student = await _context.Students
        .Include(s => s.Enrollments)
        .ThenInclude(e => e.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

The Include and ThenInclude methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. These methods are examined in detail in the Reading related data tutorial.

The AsNoTracking method improves performance in scenarios where the entities returned are not updated in the current context. `AsNoTracking` is discussed later in this tutorial.

Display enrollments

Replace the code in `Pages/Students/Details.cshtml` with the following code to display a list of enrollments. The changes are highlighted.

```
@page
@model ContosoUniversity.Pages.Students.DetailsModel

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

<h1>Details</h1>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.Enrollments)
        </dt>
        <dd class="col-sm-10">
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Student.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Student.ID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

The preceding 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 app, select the **Students** tab, and click the **Details** link for a student. The list of courses and grades for

the selected student is displayed.

Ways to read one entity

The generated code uses FirstOrDefaultAsync to read one entity. This method returns null if nothing is found; otherwise, it returns the first row found that satisfies the query filter criteria. `FirstOrDefaultAsync` is generally a better choice than the following alternatives:

- SingleOrDefaultAsync - Throws an exception if there's more than one entity that satisfies the query filter. To determine if more than one row could be returned by the query, `SingleOrDefaultAsync` tries to fetch multiple rows. This extra work is unnecessary if the query can only return one entity, as when it searches on a unique key.
- FindAsync - Finds an entity with the primary key (PK). If an entity with the PK is being tracked by the context, it's returned without a request to the database. This method is optimized to look up a single entity, but you can't call `Include` with `FindAsync`. So if related data is needed, `FirstOrDefaultAsync` is the better choice.

Route data vs. query string

The URL for the Details page is `https://localhost:<port>/Students/Details?id=1`. The entity's primary key value is in the query string. Some developers prefer to pass the key value in route data: `https://localhost:<port>/Students/Details/1`. For more information, see Update the generated code.

Update the Create page

The scaffolded `OnPostAsync` code for the Create page is vulnerable to overposting. Replace the `OnPostAsync` method in `Pages/Students/Create.cshtml.cs` with the following code.

```
public async Task<IActionResult> OnPostAsync()
{
    var emptyStudent = new Student();

    if (await TryUpdateModelAsync<Student>(
        emptyStudent,
        "student",   // Prefix for form value.
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        _context.Students.Add(emptyStudent);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

TryUpdateModelAsync

The preceding code creates a Student object and then uses posted form fields to update the Student object's properties. The TryUpdateModelAsync method:

- Uses the posted form values from the PageContext property in the PageModel.
- Updates only the properties listed (`s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate`).
- Looks for form fields with a "student" prefix. For example, `Student.FirstMidName`. It's not case sensitive.
- Uses the model binding system to convert form values from strings to the types in the `Student` model. For example, `EnrollmentDate` has to be converted to DateTime.

Run the app, and create a student entity to test the Create page.

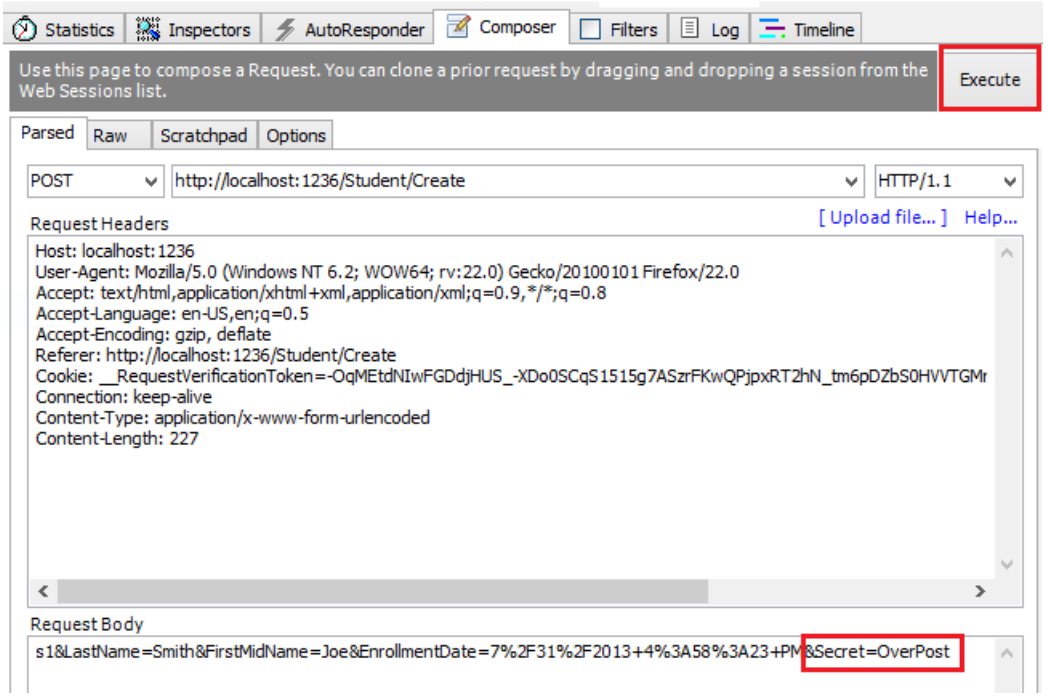
Overposting

Using `TryUpdateModel` to update fields with posted values is a security best practice because it prevents overposting. For example, suppose the Student entity includes a `Secret` property that this web page shouldn't update or add:

```
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 the app doesn't have a `Secret` field on the create or update Razor Page, a hacker could set the `Secret` value by overposting. A hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. The original code doesn't limit the fields that the model binder uses when it creates a Student instance.

Whatever value the hacker specified for the `Secret` form field is updated in the database. The following image shows the Fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" is successfully added to the `Secret` property of the inserted row. That happens even though the app designer never intended the `Secret` property to be set with the Create page.

View model

View models provide an alternative way to prevent overposting.

The application model is often called the domain model. The domain model typically contains all the properties required by the corresponding entity in the database. The view model contains only the properties needed for the UI that it is used for (for example, the Create page).

In addition to the view model, some apps use a binding model or input model to pass data between the Razor Pages page model class and the browser.

Consider the following `Student` view model:

```
using System;

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

The following code uses the `StudentVM` view model to create a new student:

```
[BindProperty]
public StudentVM StudentVM { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var entry = _context.Add(new Student());
    entry.CurrentValues.SetValues(StudentVM);
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
```

The SetValues method sets the values of this object by reading values from another PropertyValues object. `SetValues` uses property name matching. The view model type doesn't need to be related to the model type, it just needs to have properties that match.

Using `StudentVM` requires Create.cshtml be updated to use `StudentVM` rather than `Student`.

Update the Edit page

In `Pages/Students/Edit.cshtml.cs`, replace the `OnGetAsync` and `OnPostAsync` methods with the following code.

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

    Student = await _context.Students.FindAsync(id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}

public async Task<IActionResult> OnPostAsync(int id)
{
    var studentToUpdate = await _context.Students.FindAsync(id);

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

    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "student",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

The code changes are similar to the Create page with a few exceptions:

- `FirstOrDefaultAsync` has been replaced with FindAsync. When included related data is not needed, `FindAsync` is more efficient.
- `OnPostAsync` has an `id` parameter.
- The current student is fetched from the database, rather than creating an empty student.

Run the app, and test it by creating and editing a student.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database. This tracking information determines what happens when SaveChangesAsync is called. For example, when a new entity is passed to the AddAsync method, that entity's state is set to Added. When `SaveChangesAsync` is called, the database context issues a SQL INSERT command.

An entity may be in one of the following states:

- `Added`: The entity doesn't yet exist in the database. The `SaveChanges` method issues an INSERT statement.
- `Unchanged`: No changes need to be saved with this entity. An entity has this status when it's read from the database.
- `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 app, state changes are typically set automatically. An entity is read, changes are made, and the entity state is automatically changed to `Modified`. Calling `SaveChanges` generates a SQL UPDATE statement that updates only the changed properties.

In a web app, the `DbContext` that reads an entity and displays the data is disposed after a page is rendered. When a page's `OnPostAsync` method is called, a new web request is made and with a new instance of the `DbContext`. Rereading the entity in that new context simulates desktop processing.

Update the Delete page

In this section, you implement a custom error message when the call to `SaveChanges` fails.

Replace the code in `Pages/Students/Delete.cshtml.cs` with the following code. The changes are highlighted (other than cleanup of `using` statements).

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Student Student { get; set; }
        public string ErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id, bool? saveChangesError = false)
        {
            if (id == null)
            {
                return NotFound();
            }

            Student = await _context.Students
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.ID == id);

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

            if (saveChangesError.GetValueOrDefault())
            {
                ErrorMessage = "Delete failed. Try again";
            }

            return Page();
        }
```

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

            var student = await _context.Students.FindAsync(id);

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

            try
            {
                _context.Students.Remove(student);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            catch (DbUpdateException /* ex */)
            {
                //Log the error (uncomment ex variable name and write a log.)
                return RedirectToAction("./Delete",
                                        new { id, saveChangesError = true });
            }
        }
    }
}
```

The preceding code adds the optional parameter `saveChangesError` to the `OnGetAsync` method signature. `saveChangesError` indicates whether the method was called after a failure to delete the student object. The delete operation might fail because of transient network problems. Transient network errors are more likely when the database is in the cloud. The `saveChangesError` parameter is false when the Delete page `OnGetAsync` is called from the UI. When `OnGetAsync` is called by `OnPostAsync` (because the delete operation failed), the `saveChangesError` parameter is true.

The `OnPostAsync` method 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. If `Remove` fails:

- The database exception is caught.
- The Delete page's `OnGetAsync` method is called with `saveChangesError=true`.

Add an error message to the Delete Razor Page (`Pages/Students/Delete.cshtml`):

```
@page
@model ContosoUniversity.Pages.Students.DeleteModel

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

<h1>Delete</h1>

<p class="text-danger">@Model.ErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Student.ID" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Run the app and delete a student to test the Delete page.

Next steps

PREVIOUS TUTORIAL

NEXT TUTORIAL

Part 3, Razor Pages with EF Core in ASP.NET Core - Sort, Filter, Paging

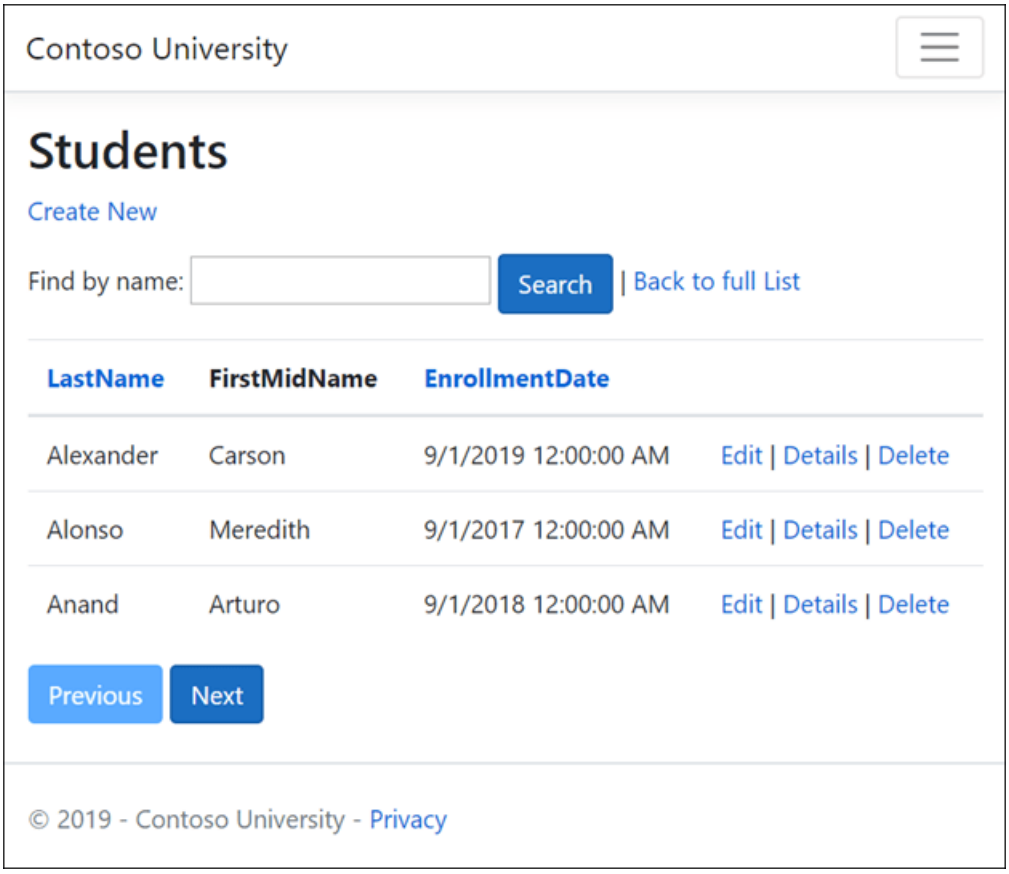
By Tom Dykstra, Jeremy Likness, and Jon P Smith

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial adds sorting, filtering, and paging functionality to the Students pages.

The following illustration shows a completed page. The column headings are clickable links to sort the column. Click a column heading repeatedly to switch between ascending and descending sort order.



Add sorting

Replace the code in `Pages/Students/Index.cshtml.cs` with the following code to add sorting.

```
public class IndexModel : PageModel
{
    private readonly SchoolContext _context;
    public IndexModel(SchoolContext context)
    {
        _context = context;
    }

    public string NameSort { get; set; }
    public string DateSort { get; set; }
    public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }

    public IList<Student> Students { get; set; }

    public async Task OnGetAsync(string sortOrder)
    {
        // using System;
        NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
        DateSort = sortOrder == "Date" ? "date_desc" : "Date";

        IQueryable<Student> studentsIQ = from s in _context.Students
                                         select s;

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

        Students = await studentsIQ.AsNoTracking().ToListAsync();
    }
}
```

The preceding code:

- Requires adding `using System;`.
- Adds properties to contain the sorting parameters.
- Changes the name of the `Student` property to `Students`.
- Replaces the code in the `OnGetAsync` method.

The `OnGetAsync` method receives a `sortOrder` parameter from the query string in the URL. The URL and query string is generated by the Anchor Tag Helper.

The `sortOrder` parameter is either `Name` or `Date`. The `sortOrder` parameter is optionally followed by `_desc` to specify descending order. The default sort order is ascending.

When the Index page is requested from the **Students** link, there's no query string. The students are displayed in ascending order by last name. Ascending order by last name is the `default` in the `switch` statement. When the user clicks a column heading link, the appropriate `sortOrder` value is provided in the query string value.

`NameSort` and `DateSort` are used by the Razor Page to configure the column heading hyperlinks with the appropriate query string values:

```
NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
DateSort = sortOrder == "Date" ? "date_desc" : "Date";
```

The code uses the C# conditional operator ?:. The `?:` operator is a ternary operator, it takes three operands. The first line specifies that when `sortOrder` is null or empty, `NameSort` is set to `name_desc`. If `sortOrder` is ***not*** null or empty, `NameSort` is set to an empty string.

These two statements enable the page to set the column heading hyperlinks as follows:

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 initializes an `IQueryable<Student>` before the switch statement, and modifies it in the switch statement:

```
IQueryable<Student> studentsIQ = from s in _context.Students
                                 select s;

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

Students = await studentsIQ.AsNoTracking().ToListAsync();
```

When an `IQueryable` is created or modified, no query is sent to the database. The query isn't executed until the `IQueryable` object is converted into a collection. `IQueryable` are converted to a collection by calling a method such as `ToListAsync`. Therefore, the `IQueryable` code results in a single query that's not executed until the following statement:

```
Students = await studentsIQ.AsNoTracking().ToListAsync();
```

`OnGetAsync` could get verbose with a large number of sortable columns. For information about an alternative way to code this functionality, see Use dynamic LINQ to simplify code in the MVC version of this tutorial series.

Add column heading hyperlinks to the Student Index page

Replace the code in `Students/Index.cshtml`, with the following code. The changes are highlighted.

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

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

<h2>Students</h2>
<p>
    <a asp-page="Create">Create New</a>
</p>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model => model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model => model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <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-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding code:

- Adds hyperlinks to the `LastName` and `EnrollmentDate` column headings.
- Uses the information in `NameSort` and `DateSort` to set up hyperlinks with the current sort order values.
- Changes the page heading from Index to Students.
- Changes `Model.Student` to `Model.Students`.

To verify that sorting works:

- Run the app and select the **Students** tab.

- Click the column headings.

Add filtering

To add filtering to the Students Index page:

- A text box and a submit button is added to the Razor Page. The text box supplies a search string on the first or last name.
- The page model is updated to use the text box value.

Update the OnGetAsync method

Replace the code in `Students/Index.cshtml.cs` with the following code to add filtering:

```
public class IndexModel : PageModel
{
    private readonly SchoolContext _context;

    public IndexModel(SchoolContext context)
    {
        _context = context;
    }

    public string NameSort { get; set; }
    public string DateSort { get; set; }
    public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }

    public IList<Student> Students { get; set; }

    public async Task OnGetAsync(string sortOrder, string searchString)
    {
        NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
        DateSort = sortOrder == "Date" ? "date_desc" : "Date";

        CurrentFilter = searchString;

        IQueryable<Student> studentsIQ = from s in _context.Students
                                         select s;
        if (!String.IsNullOrEmpty(searchString))
        {
            studentsIQ = studentsIQ.Where(s => s.LastName.Contains(searchString)
                                   || s.FirstMidName.Contains(searchString));
        }

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

        Students = await studentsIQ.AsNoTracking().ToListAsync();
    }
}
```

The preceding code:

- Adds the `searchString` parameter to the `OnGetAsync` method, and saves the parameter value in the `CurrentFilter` property. The search string value is received from a text box that's added in the next section.
- Adds to the LINQ statement a `Where` clause. The `Where` clause selects only students whose first name or last name contains the search string. The LINQ statement is executed only if there's a value to search for.

IQueryable vs. IEnumerable

The code calls the Where method on an `IQueryable` object, and the filter is processed on the server. In some scenarios, the app might be calling the `Where` method as an extension method on an in-memory collection. For example, suppose `_context.Students` changes from EF Core `DbSet` to 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 `Contains` performs a case-sensitive comparison by default. In SQL Server, `Contains` case-sensitivity is determined by the collation setting of the SQL Server instance. SQL Server defaults to case-insensitive. SQLite defaults to case-sensitive. `ToUpper` could be called to make the test explicitly case-insensitive:

```
Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())`
```

The preceding code would ensure that the filter is case-insensitive even if the `Where` method is called on an `IEnumerable` or runs on SQLite.

When `Contains` is called on an `IEnumerable` collection, the .NET Core implementation is used. When `Contains` is called on an `IQueryable` object, the database implementation is used.

Calling `Contains` on an `IQueryable` is usually preferable for performance reasons. With `IQueryable`, the filtering is done by the database server. If an `IEnumerable` is created first, all the rows have to be returned from the database server.

There's a performance penalty for calling `ToUpper`. The `ToUpper` code adds a function in the WHERE clause of the TSQL SELECT statement. The added function prevents the optimizer from using an index. Given that SQL is installed as case-insensitive, it's best to avoid the `ToUpper` call when it's not needed.

For more information, see How to use case-insensitive query with Sqlite provider.

Update the Razor page

Replace the code in `Pages/Students/Index.cshtml` to add a **Search** button.

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

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

<h2>Students</h2>

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

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-primary" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model => model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model => model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <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-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding 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. With POST, the parameters are passed in the HTTP message body and not in the URL. When HTTP GET is used, the form data is passed in the URL as query strings. Passing the data with query strings enables users to bookmark the URL. The W3C guidelines recommend that GET should be used when the action doesn't result in an update.

Test the app:

- Select the **Students** tab and enter a search string. If you're using SQLite, the filter is case-insensitive only if you implemented the optional `ToUpper` code shown earlier.
- Select **Search**.

Notice that the URL contains the search string. For example:

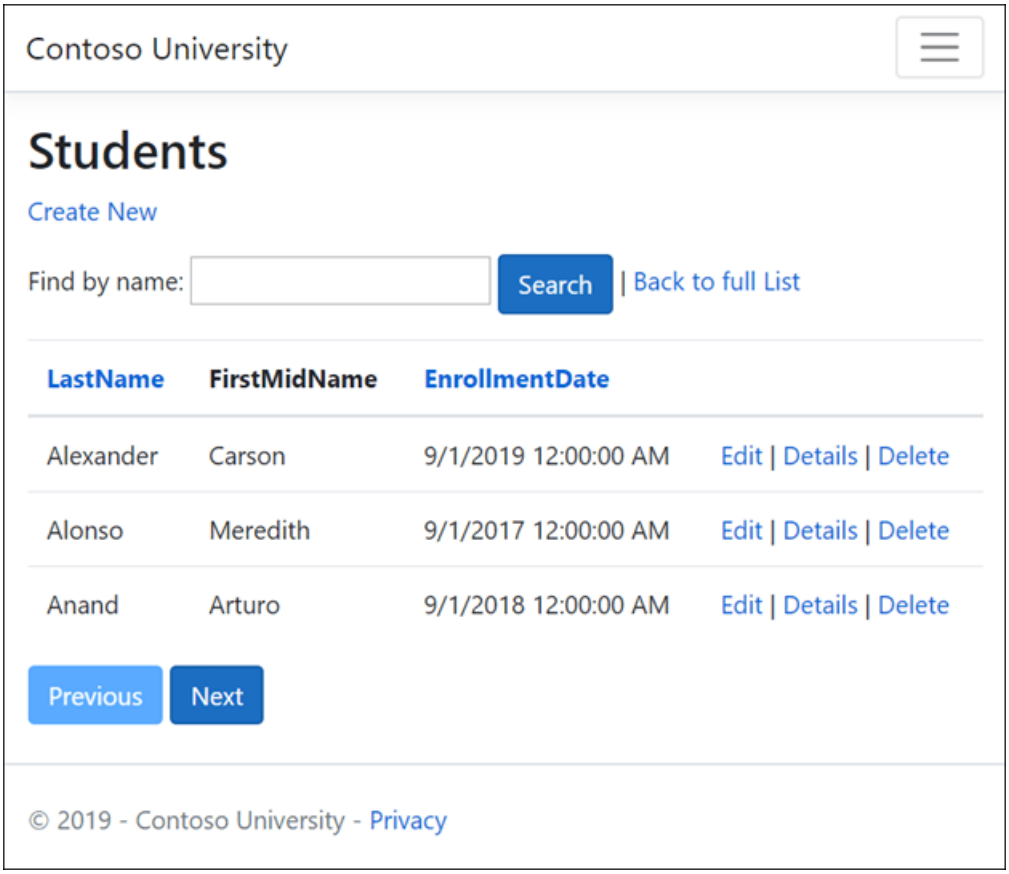
```
https://localhost:5001/Students?SearchString=an
```

If the page is bookmarked, the bookmark contains the URL to the page and the `SearchString` query string. The `method="get"` in the `form` tag is what caused the query string to be generated.

Currently, when a column heading sort link is selected, the filter value from the **Search** box is lost. The lost filter value is fixed in the next section.

Add paging

In this section, a `PaginatedList` class is created to support paging. The `PaginatedList` class uses `Skip` and `Take` statements to filter data on the server instead of retrieving all rows of the table. The following illustration shows the paging buttons.



Create the PaginatedList class

In the project folder, create `PaginatedList.cs` with the following code:

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

namespace ContosoUniversity
{
    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 => PageIndex > 1;

        public bool HasNextPage => 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 the preceding 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 returns a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` are used to enable or disable **Previous** and **Next** paging buttons.

The `CreateAsync` method is used to create the `PaginatedList<T>`. A constructor can't create the `PaginatedList<T>` object; constructors can't run asynchronous code.

Add page size to configuration

Add `PageSize` to the `appsettings.json` Configuration file:

```
{
  "PageSize": 3,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=(localdb)\\mssqllocaldb;Database=CU-
1;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

Add paging to IndexModel

Replace the code in `Students/Index.cshtml.cs` to add paging.

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class IndexModel : PageModel
    {
        private readonly SchoolContext _context;
        private readonly IConfiguration Configuration;

        public IndexModel(SchoolContext context, IConfiguration configuration)
        {
            _context = context;
            Configuration = configuration;
        }

        public string NameSort { get; set; }
        public string DateSort { get; set; }
        public string CurrentFilter { get; set; }
        public string CurrentSort { get; set; }

        public PaginatedList<Student> Students { get; set; }

        public async Task OnGetAsync(string sortOrder,
            string currentFilter, string searchString, int? pageIndex)
        {
            CurrentSort = sortOrder;
            NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
            DateSort = sortOrder == "Date" ? "date_desc" : "Date";
            if (searchString != null)
            {
                pageIndex = 1;
            }
            else
            {
                searchString = currentFilter;
            }

            CurrentFilter = searchString;
```

```
            IQueryable<Student> studentsIQ = from s in _context.Students
                                             select s;
            if (!String.IsNullOrEmpty(searchString))
            {
                studentsIQ = studentsIQ.Where(s => s.LastName.Contains(searchString)
                                       || s.FirstMidName.Contains(searchString));
            }
            switch (sortOrder)
            {
                case "name_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
                    break;
                case "Date":
                    studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                    break;
                case "date_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s => s.EnrollmentDate);
                    break;
                default:
                    studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                    break;
            }

            var pageSize = Configuration.GetValue("PageSize", 4);
            Students = await PaginatedList<Student>.CreateAsync(
                studentsIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
        }
    }
}
```

The preceding code:

- Changes the type of the `Students` property from `IList<Student>` to `PaginatedList<Student>`.
- Adds the page index, the current `sortOrder`, and the `currentFilter` to the `OnGetAsync` method signature.
- Saves the sort order in the `CurrentSort` property.
- Resets page index to 1 when there's a new search string.
- Uses the `PaginatedList` class to get Student entities.
- Sets `pageSize` to 3 from Configuration, 4 if configuration fails.

All the parameters that `OnGetAsync` receives are null when:

- The page is called from the **Students** link.
- The user hasn't clicked a paging or sorting link.

When a paging link is clicked, the page index variable contains the page number to display.

The `CurrentSort` property provides the Razor Page with the current sort order. The current sort order must be included in the paging links to keep the sort order while paging.

The `CurrentFilter` property provides the Razor Page with the current filter string. The `CurrentFilter` value:

- Must be included in the paging links in order to maintain the filter settings during paging.
- Must be restored to the text box when the page is redisplayed.

If the search string is changed while paging, the page is reset to 1. The page has to be reset to 1 because the new filter can result in different data to display. When a search value is entered and **Submit** is selected:

- The search string is changed.
- The `searchString` parameter isn't null.

The `PaginatedList.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 passed to the Razor Page.

The two question marks after `pageIndex` in the `PaginatedList.CreateAsync` call represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type. The expression `pageIndex ?? 1` returns the value of `pageIndex` if it has a value, otherwise, it returns 1.

Add paging links

Replace the code in `Students/Index.cshtml` with the following code. The changes are highlighted:

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

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

<h2>Students</h2>

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

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-primary" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <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-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

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

<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @nextDisabled">
    Next
</a>
```

The column header links use the query string to pass the current search string to the `OnGetAsync` method:

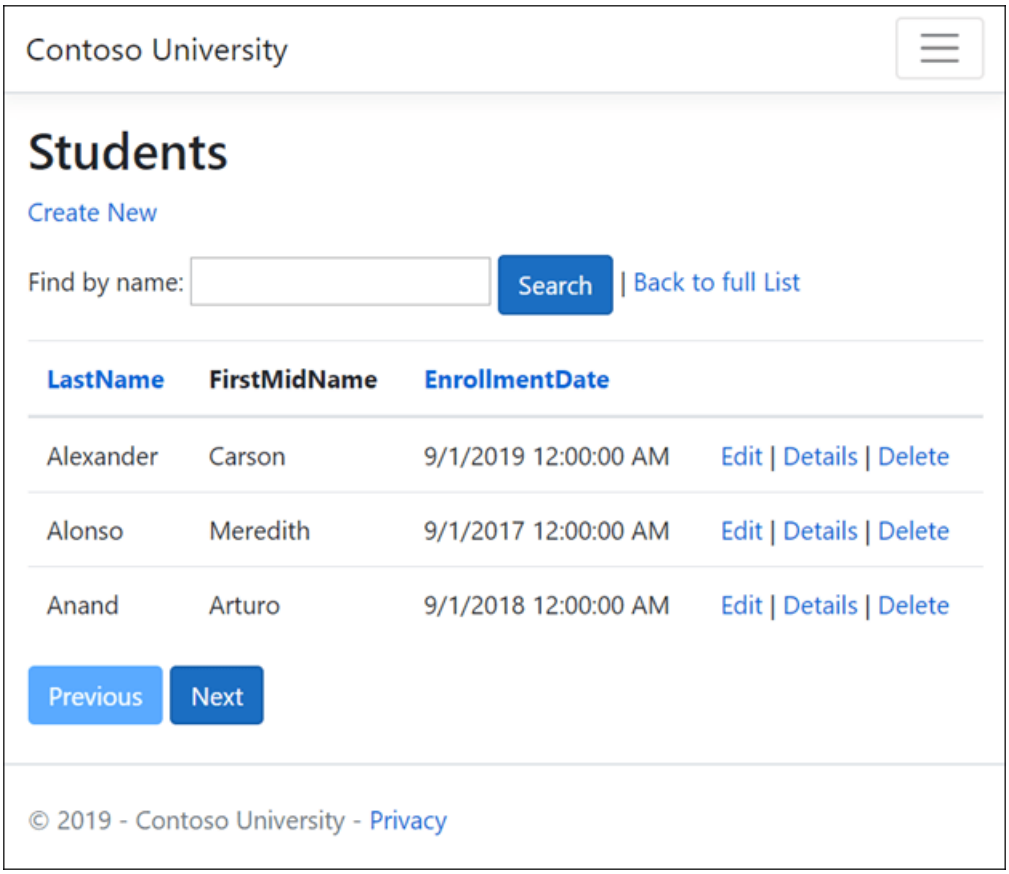
```
<a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
   asp-route-currentFilter="@Model.CurrentFilter">
    @Html.DisplayNameFor(model => model.Students[0].LastName)
</a>
```

The paging buttons are displayed by tag helpers:

```
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @nextDisabled">
    Next
</a>
```

Run the app and navigate to the students page.

- To make sure paging works, click the paging links in different sort orders.
- To verify that paging works correctly with sorting and filtering, enter a search string and try paging.



Grouping

This section creates an `About` page that displays how many students have enrolled for each enrollment date. The update uses grouping and includes the following steps:

- Create a view model for the data used by the `About` page.
- Update the `About` page to use the view model.

Create the view model

Create a *Models/SchoolViewModels* folder.

Create `SchoolViewModels/EnrollmentDateGroup.cs` with the following code:

```
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; }
    }
}
```

Create the Razor Page

Create a `Pages/About.cshtml` file with the following code:

```
@page
@model ContosoUniversity.Pages.AboutModel

@{
    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.Students)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

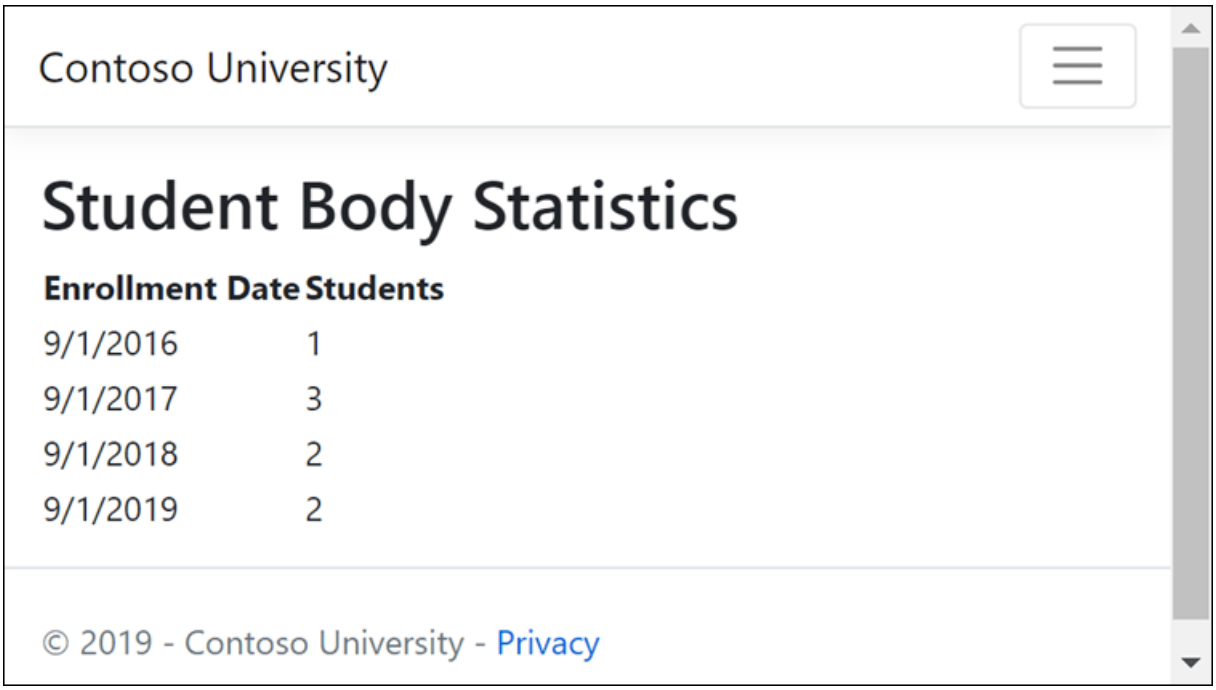
Create the page model

Update the `Pages/About.cshtml.cs` file with the following code:

```
using ContosoUniversity.Models.SchoolViewModels;
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ContosoUniversity.Models;

namespace ContosoUniversity.Pages
{
    public class AboutModel : PageModel
    {
        private readonly SchoolContext _context;

        public AboutModel(SchoolContext context)
        {
            _context = context;
        }

        public IList<EnrollmentDateGroup> Students { get; set; }

        public async Task OnGetAsync()
        {
            IQueryable<EnrollmentDateGroup> data =
                from student in _context.Students
                group student by student.EnrollmentDate into dateGroup
                select new EnrollmentDateGroup()
                {
                    EnrollmentDate = dateGroup.Key,
                    StudentCount = dateGroup.Count()
                };

            Students = 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.

Run the app and navigate to the About page. The count of students for each enrollment date is displayed in a table.



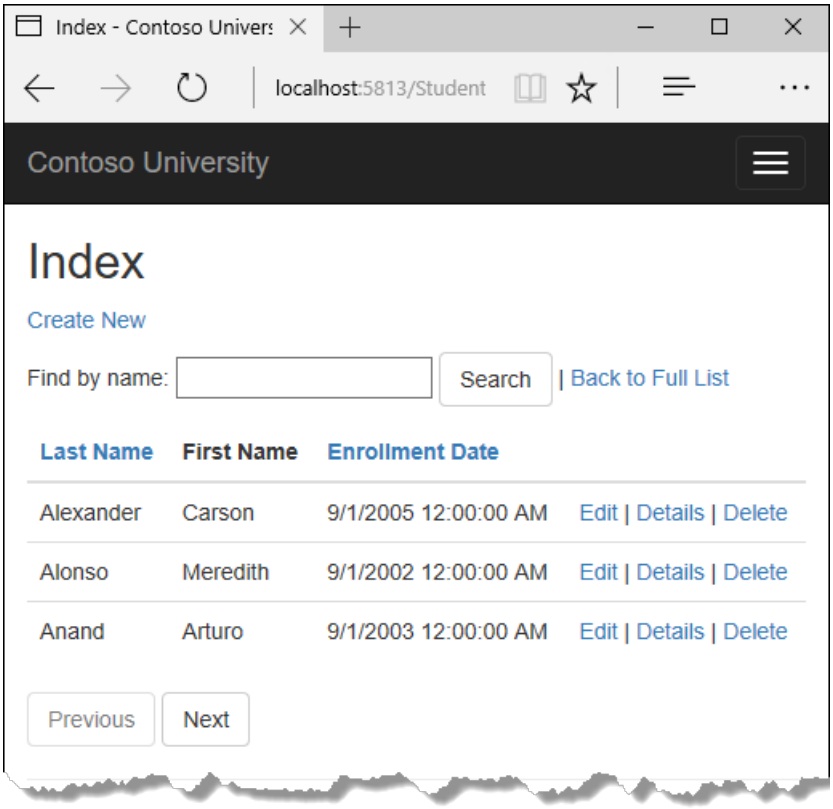
Next steps

In the next tutorial, the app uses migrations to update the data model.

PREVIOUS TUTORIAL NEXT TUTORIAL

In this tutorial, sorting, filtering, grouping, and paging, functionality is added.

The following illustration shows a completed page. The column headings are clickable links to sort the column. Clicking a column heading repeatedly switches between ascending and descending sort order.



If you run into problems you can't solve, download the completed app.

Add sorting to the Index page

Add strings to the `Students/Index.cshtml.cs` `PageModel` to contain the sorting parameters:

```
public class IndexModel : PageModel
{
    private readonly SchoolContext _context;

    public IndexModel(SchoolContext context)
    {
        _context = context;
    }

    public string NameSort { get; set; }
    public string DateSort { get; set; }
    public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }
```

Update the `Students/Index.cshtml.cs` `OnGetAsync` with the following code:

```
public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;

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

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

The preceding code receives a `sortOrder` parameter from the query string in the URL. The URL (including the query string) is generated by the Anchor Tag Helper

The `sortOrder` parameter is either "Name" or "Date." The `sortOrder` parameter is optionally followed by "_desc" to specify descending order. The default sort order is ascending.

When the Index page is requested from the **Students** link, there's no query string. The students are displayed in ascending order by last name. Ascending order by last name is the default (fall-through case) in the `switch` statement. When the user clicks a column heading link, the appropriate `sortOrder` value is provided in the query string value.

`NameSort` and `DateSort` are used by the Razor Page to configure the column heading hyperlinks with the

appropriate query string values:

```
public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;

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

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

The following code contains the C# conditional ?: operator:

```
NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
DateSort = sortOrder == "Date" ? "date_desc" : "Date";
```

The first line specifies that when `sortOrder` is null or empty, `NameSort` is set to "name_desc." If `sortOrder` is **not** null or empty, `NameSort` is set to an empty string.

The `?: operator` is also known as the ternary operator.

These two statements enable the page to set the column heading hyperlinks as follows:

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 initializes an `IQueryable<Student>` before the switch statement, and modifies it in the switch statement:

```
public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;

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

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

When an `IQueryable` is created or modified, no query is sent to the database. The query isn't executed until the `IQueryable` object is converted into a collection. `IQueryable` are converted to a collection by calling a method such as `ToListAsync`. Therefore, the `IQueryable` code results in a single query that's not executed until the following statement:

```
Student = await studentIQ.AsNoTracking().ToListAsync();
```

`OnGetAsync` could get verbose with a large number of sortable columns.

Add column heading hyperlinks to the Student Index page

Replace the code in `Students/Index.cshtml`, with the following highlighted code:

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

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

<h2>Index</h2>
<p>
    <a asp-page="Create">Create New</a>
</p>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model => model.Student[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Student[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model => model.Student[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Student)
        {
            <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-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding code:

- Adds hyperlinks to the `LastName` and `EnrollmentDate` column headings.
- Uses the information in `NameSort` and `DateSort` to set up hyperlinks with the current sort order values.

To verify that sorting works:

- Run the app and select the **Students** tab.
- Click **Last Name**.
- Click **Enrollment Date**.

To get a better understanding of the code:

- In `Students/Index.cshtml.cs`, set a breakpoint on `switch (sortOrder)`.
- Add a watch for `NameSort` and `DateSort`.
- In `Students/Index.cshtml`, set a breakpoint on `@Html.DisplayNameFor(model => model.Student[0].LastName)`.

Step through the debugger.

Add a Search Box to the Students Index page

To add filtering to the Students Index page:

- A text box and a submit button is added to the Razor Page. The text box supplies a search string on the first or last name.
- The page model is updated to use the text box value.

Add filtering functionality to the Index method

Update the `Students/Index.cshtml.cs` `OnGetAsync` with the following code:

```
public async Task OnGetAsync(string sortOrder, string searchString)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";
    CurrentFilter = searchString;

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        studentIQ = studentIQ.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }

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

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

The preceding code:

- Adds the `searchString` parameter to the `OnGetAsync` method. The search string value is received from a text box that's added in the next section.
- Added to the LINQ statement a `Where` clause. The `Where` clause selects only students whose first name or last name contains the search string. The LINQ statement is executed only if there's a value to search for.

Note: The preceding code calls the `Where` method on an `IQueryable` object, and the filter is processed on the server. In some scenarios, the app might be calling the `Where` method as an extension method on an in-

memory collection. For example, suppose `_context.Students` changes from EF Core `DbSet` to 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 `Contains` performs a case-sensitive comparison by default. In SQL Server, `Contains` case-sensitivity is determined by the collation setting of the SQL Server instance. SQL Server defaults to case-insensitive. `ToUpper` could be called to make the test explicitly case-insensitive:

```
Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())
```

The preceding code would ensure that results are case-insensitive if the code changes to use `IEnumerable`. When `Contains` is called on an `IEnumerable` collection, the .NET Core implementation is used. When `Contains` is called on an `IQueryable` object, the database implementation is used. Returning an `IEnumerable` from a repository can have a significant performance penalty:

1. All the rows are returned from the DB server.
2. The filter is applied to all the returned rows in the application.

There's a performance penalty for calling `ToUpper`. The `ToUpper` code adds a function in the WHERE clause of the TSQL SELECT statement. The added function prevents the optimizer from using an index. Given that SQL is installed as case-insensitive, it's best to avoid the `ToUpper` call when it's not needed.

Add a Search Box to the Student Index page

In `Pages/Students/Index.cshtml`, add the following highlighted code to create a **Search** button and assorted chrome.

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

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

<h2>Index</h2>

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

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
```

The preceding 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. With POST, the parameters are passed in the HTTP message body and not in the URL. When HTTP GET is used, the form data is passed in the URL as query strings. Passing the data with query strings enables users to bookmark the URL. The W3C guidelines recommend that GET should be used when the action doesn't result in an update.

Test the app:

- Select the **Students** tab and enter a search string.
- Select **Search**.

Notice that the URL contains the search string.

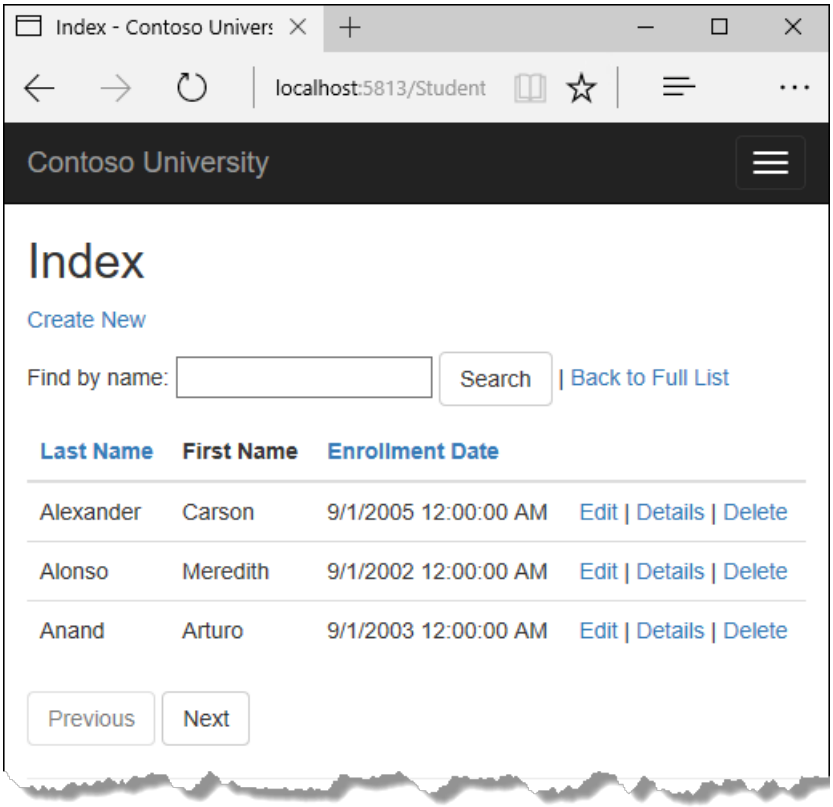
```
http://localhost:5000/Students?SearchString=an
```

If the page is bookmarked, the bookmark contains the URL to the page and the `SearchString` query string. The `method="get"` in the `form` tag is what caused the query string to be generated.

Currently, when a column heading sort link is selected, the filter value from the **Search** box is lost. The lost filter value is fixed in the next section.

Add paging functionality to the Students Index page

In this section, a `PaginatedList` class is created to support paging. The `PaginatedList` class uses `Skip` and `Take` statements to filter data on the server instead of retrieving all rows of the table. The following illustration shows the paging buttons.



In the project folder, create `PaginatedList.cs` with the following code:

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

namespace ContosoUniversity
{
    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 => PageIndex > 1;

        public bool HasNextPage => 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 the preceding 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 returns a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` are used to enable or disable **Previous** and **Next** paging buttons.

The `CreateAsync` method is used to create the `PaginatedList<T>`. A constructor can't create the `PaginatedList<T>` object, constructors can't run asynchronous code.

Add paging functionality to the Index method

In `Students/Index.cshtml.cs`, update the type of `Student` from `IList<Student>` to `PaginatedList<Student>`:

```
public PaginatedList<Student> Student { get; set; }
```

Update the `Students/Index.cshtml.cs` `OnGetAsync` with the following code:

```
public async Task OnGetAsync(string sortOrder,
    string currentFilter, string searchString, int? pageIndex)
{
    CurrentSort = sortOrder;
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";
    if (searchString != null)
    {
        pageIndex = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    CurrentFilter = searchString;

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        studentIQ = studentIQ.Where(s => s.LastName.Contains(searchString)
                                || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    Student = await PaginatedList<Student>.CreateAsync(
        studentIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
}
```

The preceding code adds the page index, the current `sortOrder`, and the `currentFilter` to the method signature.

```
public async Task OnGetAsync(string sortOrder,
    string currentFilter, string searchString, int? pageIndex)
```

All the parameters are null when:

- The page is called from the **Students** link.
- The user hasn't clicked a paging or sorting link.

When a paging link is clicked, the page index variable contains the page number to display.

`CurrentSort` provides the Razor Page with the current sort order. The current sort order must be included in the paging links to keep the sort order while paging.

`CurrentFilter` provides the Razor Page with the current filter string. The `CurrentFilter` value:

- Must be included in the paging links in order to maintain the filter settings during paging.
- Must be restored to the text box when the page is redisplayed.

If the search string is changed while paging, the page is reset to 1. The page has to be reset to 1 because the new filter can result in different data to display. When a search value is entered and **Submit** is selected:

- The search string is changed.
- The `searchString` parameter isn't null.

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

The `PaginatedList.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 passed to the Razor Page.

```
Student = await PaginatedList<Student>.CreateAsync(
    studentIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
```

The two question marks in `PaginatedList.CreateAsync` represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type. The expression `(pageIndex ?? 1)` means return the value of `pageIndex` if it has a value. If `pageIndex` doesn't have a value, return 1.

Add paging links to the student Razor Page

Update the markup in `Students/Index.cshtml`. The changes are highlighted:

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

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

<h2>Index</h2>

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

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
```

```
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Student[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Student[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Student[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Student)
        {
            <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-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

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

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

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

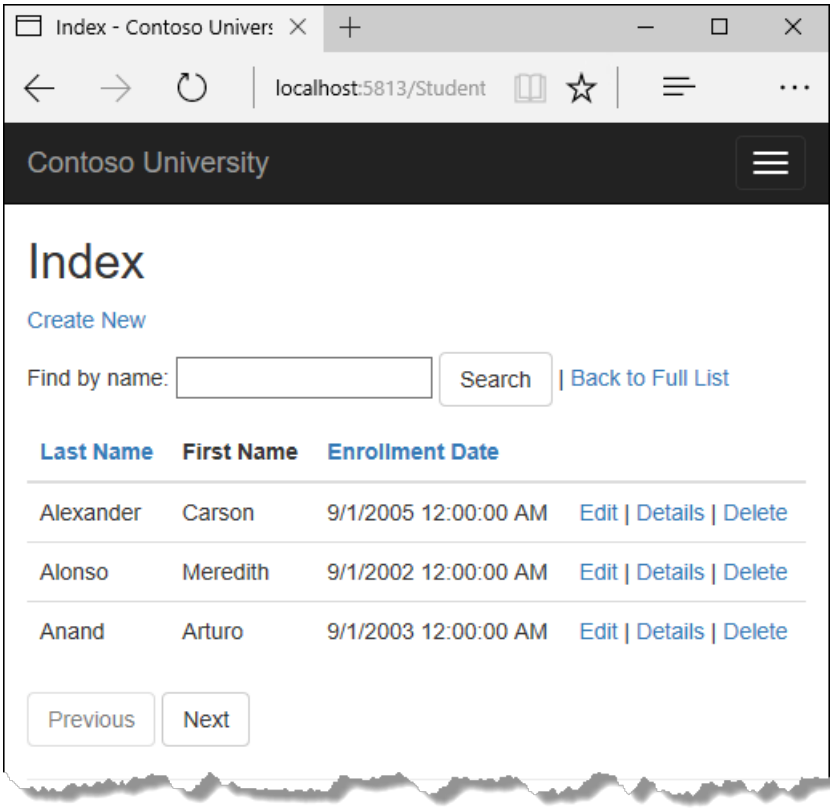
```
<a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
   asp-route-currentFilter="@Model.CurrentFilter">
    @Html.DisplayNameFor(model => model.Student[0].LastName)
</a>
```

The paging buttons are displayed by tag helpers:

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

Run the app and navigate to the students page.

- To make sure paging works, click the paging links in different sort orders.
- To verify that paging works correctly with sorting and filtering, enter a search string and try paging.



To get a better understanding of the code:

- In `Students/Index.cshtml.cs`, set a breakpoint on `switch (sortOrder)`.
- Add a watch for `NameSort`, `DateSort`, `CurrentSort`, and `Model.Student.PageIndex`.
- In `Students/Index.cshtml`, set a breakpoint on `@Html.DisplayNameFor(model => model.Student[0].LastName)`.

Step through the debugger.

Update the About page to show student statistics

In this step, `Pages/About.cshtml` is updated to display how many students have enrolled for each enrollment date. The update uses grouping and includes the following steps:

- Create a view model for the data used by the **About** Page.
- Update the About page to use the view model.

Create the view model

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

In the *SchoolViewModels* folder, add a `EnrollmentDateGroup.cs` with the following code:

```
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; }
    }
}
```

Update the About page model

The web templates in ASP.NET Core 2.2 do not include the About page. If you are using ASP.NET Core 2.2, create the About Razor Page.

Update the `Pages/About.cshtml.cs` file with the following code:

```
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ContosoUniversity.Models;

namespace ContosoUniversity.Pages
{
    public class AboutModel : PageModel
    {
        private readonly SchoolContext _context;

        public AboutModel(SchoolContext context)
        {
            _context = context;
        }

        public IList<EnrollmentDateGroup> Student { get; set; }

        public async Task OnGetAsync()
        {
            IQueryable<EnrollmentDateGroup> data =
                from student in _context.Student
                group student by student.EnrollmentDate into dateGroup
                select new EnrollmentDateGroup()
                {
                    EnrollmentDate = dateGroup.Key,
                    StudentCount = dateGroup.Count()
                };

            Student = 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.

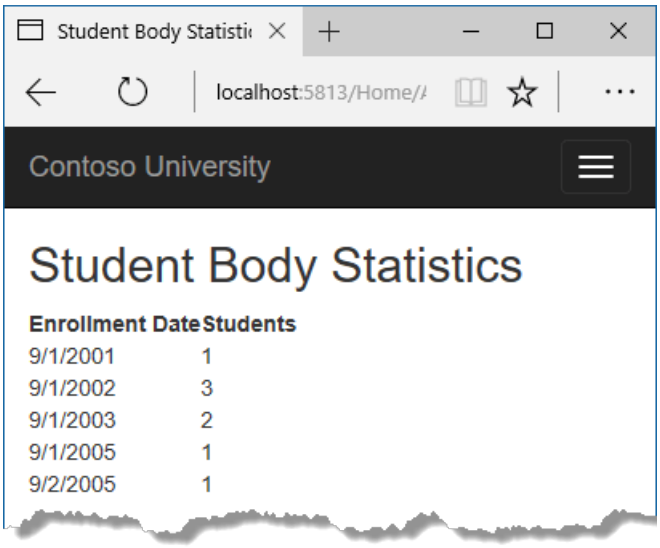
Modify the About Razor Page

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

```
@page
@model ContosoUniversity.Pages.AboutModel

@{
    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.Student)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Run the app and navigate to the About page. The count of students for each enrollment date is displayed in a table.

If you run into problems you can't solve, download the completed app for this stage.



Additional resources

- Debugging ASP.NET Core 2.x source
- YouTube version of this tutorial

In the next tutorial, the app uses migrations to update the data model.

Part 4, Razor Pages with EF Core migrations in ASP.NET Core

By Tom Dykstra, Jon P Smith, and Rick Anderson

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial introduces the EF Core migrations feature for managing data model changes.

When a new app is developed, the data model changes frequently. Each time the model changes, the model gets out of sync with the database. This tutorial series started by configuring the Entity Framework to create the database if it doesn't exist. Each time the data model changes, the database needs to be dropped. The next time the app runs, the call to `EnsureCreated` re-creates the database to match the new data model. The `DbInitializer` class then runs to seed the new database.

This approach to keeping the DB in sync with the data model works well until the app needs to be deployed to production. When the app is running in production, it's usually storing data that needs to be maintained. The app can't start with a test DB each time a change is made (such as adding a new column). The EF Core Migrations feature solves this problem by enabling EF Core to update the DB schema instead of creating a new database.

Rather than dropping and recreating the database when the data model changes, migrations updates the schema and retains existing data.

> **NOTE**
>
> **SQLite limitations**
>
> This tutorial uses the Entity Framework Core migrations feature where possible. Migrations updates the database schema to match changes in the data model. However, migrations only does the kinds of changes that the database engine supports, and SQLite's schema change capabilities are limited. For example, adding a column is supported, but removing a column is not supported. If a migration is created to remove a column, the `ef migrations add` command succeeds but the `ef database update` command fails.
>
> The workaround for the SQLite limitations is to manually write migrations code to perform a table rebuild when something in the table changes. The code goes in the `Up` and `Down` methods for a migration and involves:
>
> - Creating a new table.
> - Copying data from the old table to the new table.
> - Dropping the old table.
> - Renaming the new table.
>
> Writing database-specific code of this type is outside the scope of this tutorial. Instead, this tutorial drops and re-creates the database whenever an attempt to apply a migration would fail. For more information, see the following resources:
>
> - SQLite EF Core Database Provider Limitations
> - Customize migration code
> - Data seeding
> - SQLite ALTER TABLE statement

Drop the database

- Visual Studio
- Visual Studio Code

Use **SQL Server Object Explorer** (SSOX) to delete the database, or run the following command in the **Package Manager Console** (PMC):

```
Drop-Database
```

Create an initial migration

- Visual Studio
- Visual Studio Code

Run the following commands in the PMC:

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

Remove EnsureCreated

This tutorial series started by using EnsureCreated. `EnsureCreated` doesn't create a migrations history table and so can't be used with migrations. It's designed for testing or rapid prototyping where the database is dropped and re-created frequently.

From this point forward, the tutorials will use migrations.

In `Program.cs`, delete the following line:

```
context.Database.EnsureCreated();
```

Run the app and verify that the database is seeded.

Up and Down methods

The EF Core `migrations add` command generated code to create the database. This migrations code is in the `Migrations\<timestamp>_InitialCreate.cs` file. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets. The `Down` method deletes them, as shown in the following example:

```
using System;
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Migrations;

namespace ContosoUniversity.Migrations
{
    public partial class InitialCreate : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Course",
                columns: table => new
                {
                    CourseID = table.Column<int>(nullable: false),
                    Title = table.Column<string>(nullable: true),
                    Credits = table.Column<int>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Course", x => x.CourseID);
                });

            migrationBuilder.CreateTable(
                name: "Student",
                columns: table => new
                {
                    ID = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    LastName = table.Column<string>(nullable: true),
                    FirstMidName = table.Column<string>(nullable: true),
                    EnrollmentDate = table.Column<DateTime>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Student", x => x.ID);
                });

            migrationBuilder.CreateTable(
                name: "Enrollment",
                columns: table => new
                {
                    EnrollmentID = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    CourseID = table.Column<int>(nullable: false),
                    StudentID = table.Column<int>(nullable: false),
                    Grade = table.Column<int>(nullable: true)
                },
                constraints: table =>
```

```
                {
                    table.PrimaryKey("PK_Enrollment", x => x.EnrollmentID);
                    table.ForeignKey(
                        name: "FK_Enrollment_Course_CourseID",
                        column: x => x.CourseID,
                        principalTable: "Course",
                        principalColumn: "CourseID",
                        onDelete: ReferentialAction.Cascade);
                    table.ForeignKey(
                        name: "FK_Enrollment_Student_StudentID",
                        column: x => x.StudentID,
                        principalTable: "Student",
                        principalColumn: "ID",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateIndex(
                name: "IX_Enrollment_CourseID",
                table: "Enrollment",
                column: "CourseID");

            migrationBuilder.CreateIndex(
                name: "IX_Enrollment_StudentID",
                table: "Enrollment",
                column: "StudentID");
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Enrollment");

            migrationBuilder.DropTable(
                name: "Course");

            migrationBuilder.DropTable(
                name: "Student");
        }
    }
}
```

The preceding code is for the initial migration. The code:

- Was generated by the `migrations add InitialCreate` command.
- Is executed by the `database update` command.
- Creates a database for the data model specified by the database context class.

The migration name parameter (`InitialCreate` in the example) is used for the file name. The migration name can be any valid file name. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, a migration that added a department table might be called "AddDepartmentTable."

The migrations history table

- Use SSOX or SQLite tool to inspect the database.
- Notice the addition of an `__EFMigrationsHistory` table. The `__EFMigrationsHistory` table keeps track of which migrations have been applied to the database.
- View the data in the `__EFMigrationsHistory` table. It shows one row for the first migration.

The data model snapshot

Migrations creates a *snapshot* of the current data model in `Migrations/SchoolContextModelSnapshot.cs`. When add a migration is added, EF determines what changed by comparing the current data model to the snapshot

file.

Because the snapshot file tracks the state of the data model, a migration cannot be deleted by deleting the `<timestamp>_<migrationname>.cs` file. To back out the most recent migration, use the `migrations remove` command. `migrations remove` deletes the migration and ensures the snapshot is correctly reset. For more information, see dotnet ef migrations remove.

See Resetting all migrations to remove all migrations.

Applying migrations in production

We recommend that production apps **not** call Database.Migrate at application startup. `Migrate` shouldn't be called from an app that is deployed to a server farm. If the app is scaled out to multiple server instances, it's hard to ensure database schema updates don't happen from multiple servers or conflict with read/write access.

Database migration should be done as part of deployment, and in a controlled way. Production database migration approaches include:

- Using migrations to create SQL scripts and using the SQL scripts in deployment.
- Running `dotnet ef database update` from a controlled environment.

Troubleshooting

If the app uses SQL Server LocalDB and displays the following exception:

```
SqlException: Cannot open database "ContosoUniversity" requested by the login.
The login failed.
Login failed for user 'user name'.
```

The solution may be to run `dotnet ef database update` at a command prompt.

Additional resources

- EF Core CLI.
- dotnet ef migrations CLI commands
- Package Manager Console (Visual Studio)

Next steps

The next tutorial builds out the data model, adding entity properties and new entities.

PREVIOUS TUTORIAL | NEXT TUTORIAL

In this tutorial, the EF Core migrations feature for managing data model changes is used.

If you run into problems you can't solve, download the completed app.

When a new app is developed, the data model changes frequently. Each time the model changes, the model gets out of sync with the database. This tutorial started by configuring the Entity Framework to create the database if it doesn't exist. Each time the data model changes:

- The DB is dropped.
- EF creates a new one that matches the model.

- The app seeds the DB with test data.

This approach to keeping the DB in sync with the data model works well until the app needs to be deployed to production. When the app is running in production, it's usually storing data that needs to be maintained. The app can't start with a test DB each time a change is made (such as adding a new column). The EF Core Migrations feature solves this problem by enabling EF Core to update the DB schema instead of creating a new DB.

Rather than dropping and recreating the DB when the data model changes, migrations updates the schema and retains existing data.

Drop the database

Use **SQL Server Object Explorer** (SSOX) or the `database drop` command:

- Visual Studio
- Visual Studio Code

In the **Package Manager Console** (PMC), run the following command:

```
Drop-Database
```

Run `Get-Help about_EntityFrameworkCore` from the PMC to get help information.

Create an initial migration and update the DB

Build the project and create the first migration.

- Visual Studio
- Visual Studio Code

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

Examine the Up and Down methods

The EF Core `migrations add` command generated code to create the database. This migrations code is in the `Migrations\<timestamp>_InitialCreate.cs` file. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets. The `Down` method deletes them, as shown in the following example:

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(nullable: false),
                Title = table.Column<string>(nullable: true),
                Credits = table.Column<int>(nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });

        migrationBuilder.CreateTable(
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Enrollment");

        migrationBuilder.DropTable(
            name: "Course");

        migrationBuilder.DropTable(
            name: "Student");
    }
}
```

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

The preceding code is for the initial migration. That code was created when the `migrations add InitialCreate` command was run. The migration name parameter ("InitialCreate" in the example) is used for the file name. The migration name can be any valid file name. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, a migration that added a department table might be called "AddDepartmentTable."

If the initial migration is created and the DB exists:

- The DB creation code is generated.
- The DB creation code doesn't need to run because the DB already matches the data model. If the DB creation code is run, it doesn't make any changes because the DB already matches the data model.

When the app is deployed to a new environment, the DB creation code must be run to create the DB.

Previously the DB was dropped and doesn't exist, so migrations creates the new DB.

The data model snapshot

Migrations create a *snapshot* of the current database schema in `Migrations/SchoolContextModelSnapshot.cs`. When you add a migration, EF determines what changed by comparing the data model to the snapshot file.

To delete a migration, use the following command:

- Visual Studio
- Visual Studio Code

Remove-Migration

The remove migrations command deletes the migration and ensures the snapshot is correctly reset.

Remove EnsureCreated and test the app

For early development, `EnsureCreated` was used. In this tutorial, migrations are used. `EnsureCreated` has the following limitations:

- Bypasses migrations and creates the DB and schema.
- Doesn't create a migrations table.
- Can *not* be used with migrations.
- Is designed for testing or rapid prototyping where the DB is dropped and re-created frequently.

Remove `EnsureCreated`:

```
context.Database.EnsureCreated();
```

Run the app and verify the DB is seeded.

Inspect the database

Use **SQL Server Object Explorer** to inspect the DB. Notice the addition of an `__EFMigrationsHistory` table. The `__EFMigrationsHistory` table keeps track of which migrations have been applied to the DB. View the data in the `__EFMigrationsHistory` table, it shows one row for the first migration. The last log in the preceding CLI output example shows the INSERT statement that creates this row.

Run the app and verify that everything works.

Applying migrations in production

We recommend production apps should **not** call Database.Migrate at application startup. `Migrate` shouldn't be called from an app in server farm. For example, if the app has been cloud deployed with scale-out (multiple instances of the app are running).

Database migration should be done as part of deployment, and in a controlled way. Production database migration approaches include:

- Using migrations to create SQL scripts and using the SQL scripts in deployment.
- Running `dotnet ef database update` from a controlled environment.

EF Core uses the `__MigrationsHistory` table to see if any migrations need to run. If the DB is up-to-date, no migration is run.

Troubleshooting

Download the completed app.

The app generates the following exception:

```
SqlException: Cannot open database "ContosoUniversity" requested by the login.
The login failed.
Login failed for user 'user name'.
```

Solution: Run `dotnet ef database update`

Additional resources

- YouTube version of this tutorial
- .NET Core CLI.

- Package Manager Console (Visual Studio)

PREVIOUS NEXT

Part 5, Razor Pages with EF Core in ASP.NET Core - Data Model

By Tom Dykstra, Jeremy Likness, and Jon P Smith

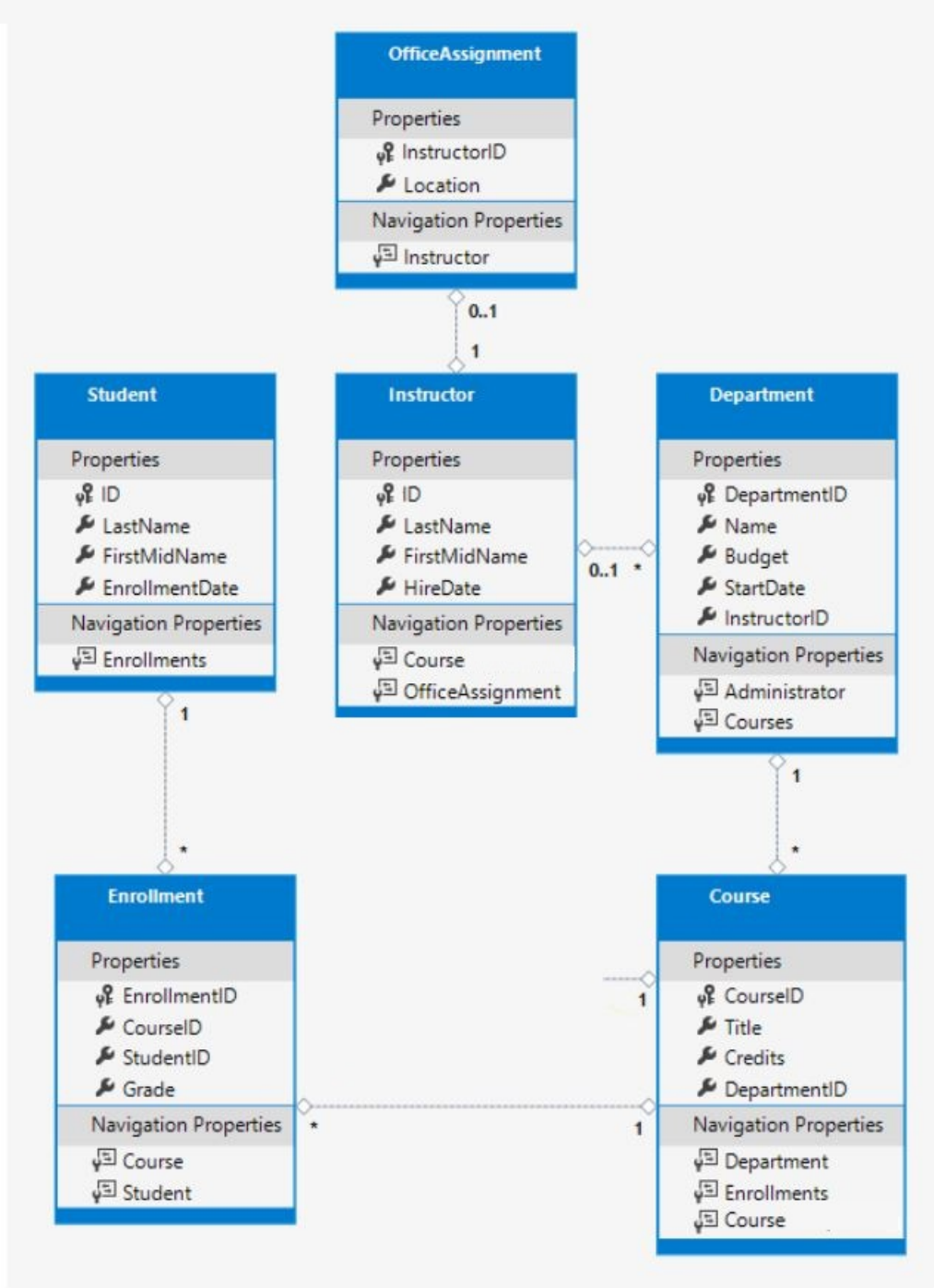
The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

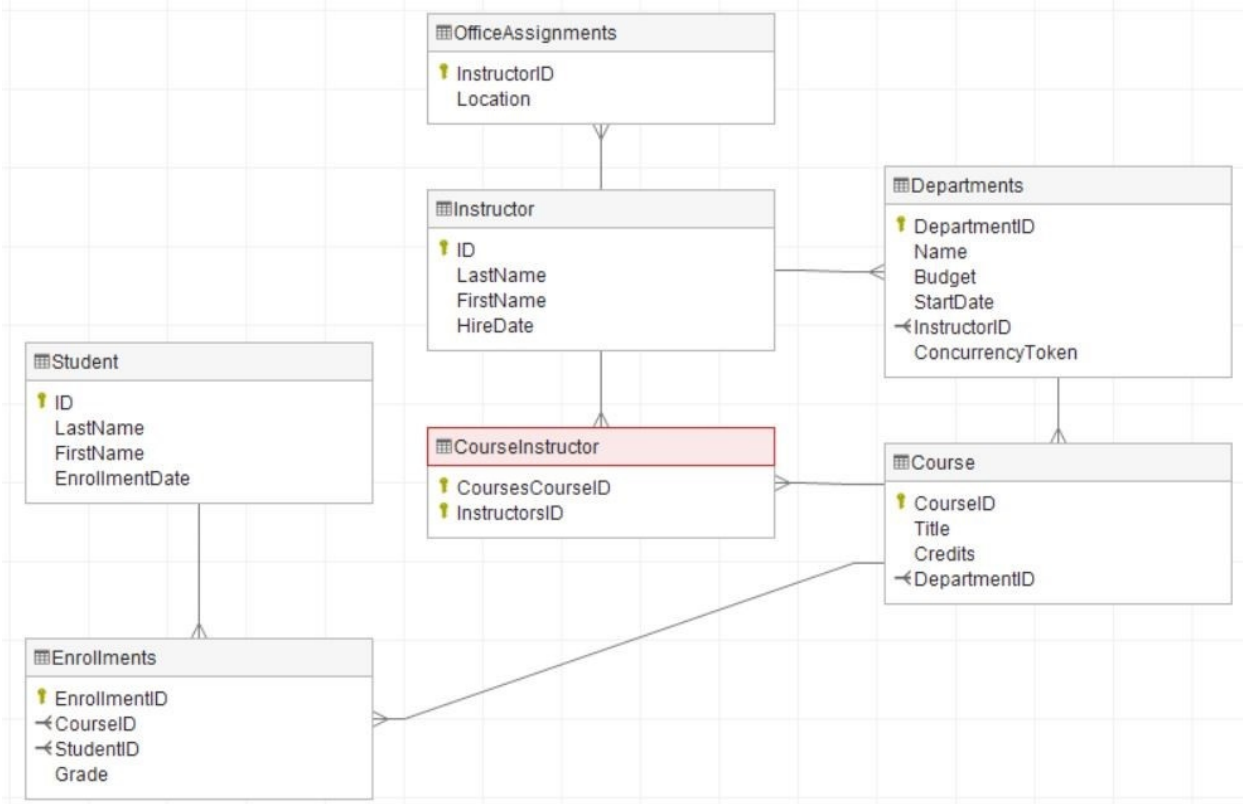
The previous tutorials worked with a basic data model that was composed of three entities. In this tutorial:

- More entities and relationships are added.
- The data model is customized by specifying formatting, validation, and database mapping rules.

The completed data model is shown in the following illustration:



The following database diagram was made with Dataedo:



To create a database diagram with Dataedo:

- Deploy the app to Azure
- Download and install Dataedo on your computer.
- Follow the instructions Generate documentation for Azure SQL Database in 5 minutes

In the preceding Dataedo diagram, the `CourseInstructor` is a join table created by Entity Framework. For more information, see Many-to-many

The Student entity

Replace the code in `Models/Student.cs` with the following code:

```
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 preceding code adds a `FullName` property and adds the following attributes to existing properties:

- `[DataType]`
- `[DisplayFormat]`
- `[StringLength]`
- `[Column]`
- `[Required]`
- `[Display]`

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. `FullName` can't be set, so it has only a get accessor. No `FullName` column is created in the database.

The DataType attribute

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

For student enrollment dates, all of the pages currently display the time of day along with the date, although only the date is relevant. By using data annotation attributes, you can make one code change that will fix the display format in every page that shows the data.

The DataType attribute specifies a data type that's more specific than the database intrinsic type. In this case only the date should be displayed, not the date and time. The DataType Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, etc. The `DataType` attribute can also enable the app

to automatically provide type-specific features. For example:

- The `mailto:` link is automatically created for `DataType.EmailAddress`.
- The date selector is provided for `DataType.Date` in most browsers.

The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes. The `DataType` attributes don't provide validation.

The DisplayFormat attribute

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

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the date 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. The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied to the edit UI. Some fields shouldn't use `ApplyFormatInEditMode`. For example, the currency symbol should generally not be displayed in an edit text box.

The `DisplayFormat` attribute can be used by itself. It's generally a good idea to use the `DataType` attribute with the `DisplayFormat` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen. The `DataType` attribute provides the following benefits that are not available in `DisplayFormat`:

- The browser can enable HTML5 features. For example, show a calendar control, the locale-appropriate currency symbol, email links, and client-side input validation.
- By default, the browser renders data using the correct format based on the locale.

For more information, see the <input> Tag Helper documentation.

The StringLength attribute

```
[StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
```

Data validation rules and validation error messages can be specified with attributes. The StringLength attribute specifies the minimum and maximum length of characters that are allowed in a data field. The code shown limits names to no more than 50 characters. An example that sets the minimum string length is shown later.

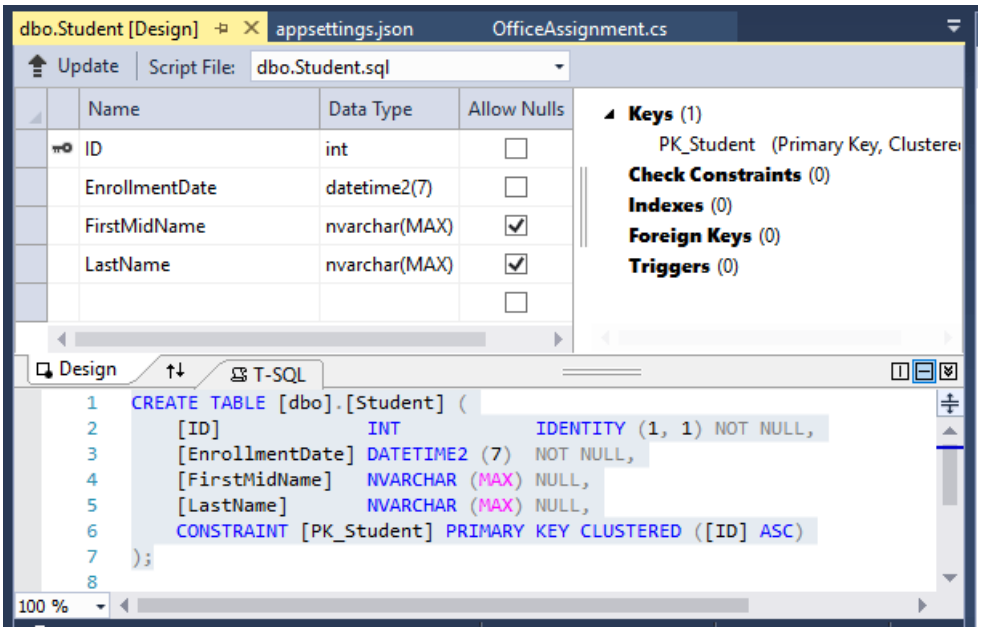
The `StringLength` attribute also provides client-side and server-side validation. The minimum value has no impact on the database schema.

The `StringLength` attribute doesn't prevent a user from entering white space for a name. The RegularExpression attribute can be used 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:

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

- Visual Studio
- Visual Studio Code

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



The preceding image shows the schema for the `Student` table. The name fields have type `nvarchar(MAX)`. When a migration is created and applied later in this tutorial, the name fields become `nvarchar(50)` as a result of the string length attributes.

The Column attribute

```
[Column("FirstName")]
public string FirstMidName { get; set; }
```

Attributes can control how classes and properties are mapped to the database. In the `Student` model, the `Column` attribute is used to map the name of the `FirstMidName` property to "FirstName" in the database.

When the database is created, property names on the model are used for column names (except when the `Column` attribute is used). The `Student` model uses `FirstMidName` for the first-name field because the field might also contain a middle name.

With the `[Column]` attribute, `Student.FirstMidName` in the data model maps to the `FirstName` column of the `Student` table. The addition of the `Column` attribute changes the model backing the `SchoolContext`. The model backing the `SchoolContext` no longer matches the database. That discrepancy will be resolved by adding a migration later in this tutorial.

The Required attribute

```
[Required]
```

The `Required` attribute makes the name properties required fields. The `Required` attribute isn't needed for non-nullable types such as value types (for example, `DateTime`, `int`, and `double`). Types that can't be null are automatically treated as required fields.

The `Required` attribute must be used with `MinimumLength` for the `MinimumLength` to be enforced.

```
[Display(Name = "Last Name")]
[Required]
[StringLength(50, MinimumLength=2)]
public string LastName { get; set; }
```

`MinimumLength` and `Required` allow whitespace to satisfy the validation. Use the `RegularExpression` attribute for

full control over the string.

The Display attribute

```
[Display(Name = "Last Name")]
```

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date." The default captions had no space dividing the words, for example "Lastname."

Create a migration

Run the app and go to the Students page. An exception is thrown. The `[Column]` attribute causes EF to expect to find a column named `FirstName`, but the column name in the database is still `FirstMidName`.

- Visual Studio
- Visual Studio Code

The error message is similar to the following example:

```
SqlException: Invalid column name 'FirstName'.
There are pending model changes
Pending model changes are detected in the following:

SchoolContext
```

- In the PMC, enter the following commands to create a new migration and update the database:

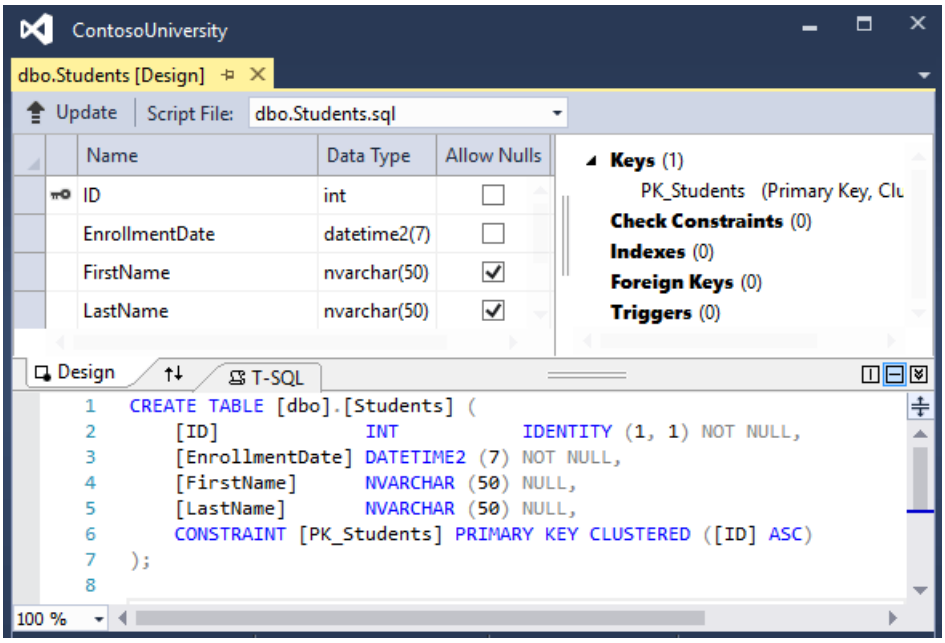
  ```
  Add-Migration ColumnFirstName
  Update-Database
  ```

  The first of these commands generates the following warning message:

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

  The warning is generated because the name fields are now limited to 50 characters. If a name in the database had more than 50 characters, the 51 to last character would be lost.

- Open the Student table in SSOX:



Before the migration was applied, the name columns were of type nvarchar(MAX). The name columns are now `nvarchar(50)`. The column name has changed from `FirstMidName` to `FirstName`.

- Run the app and go to the Students page.
- Notice that times are not input or displayed along with dates.
- Select **Create New**, and try to enter a name longer than 50 characters.

> **NOTE**
>
> In the following sections, building the app at some stages generates compiler errors. The instructions specify when to build the app.

The Instructor Entity

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

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

Multiple attributes can be on one line. The `HireDate` attributes could be written as follows:

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

Navigation properties

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

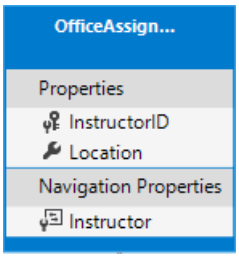
An instructor can teach any number of courses, so `Courses` is defined as a collection.

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

An instructor can have at most one office, so the `OfficeAssignment` property holds a single `OfficeAssignment` entity. `OfficeAssignment` is null if no office is assigned.

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

The OfficeAssignment entity



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

```
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

The `[Key]` attribute is used to identify a property as the primary key (PK) when the property name is something other than `classnameID` or `ID`.

There's a one-to-zero-or-one relationship between the `Instructor` and `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to. The `OfficeAssignment` PK is also its foreign key (FK) to the `Instructor` entity. A one-to-zero-or-one relationship occurs when a PK in one table is both a PK and a FK in another table.

EF Core can't automatically recognize `InstructorID` as the PK of `OfficeAssignment` because `InstructorID` doesn't follow the ID or classnameID naming convention. Therefore, the `Key` attribute is used to identify `InstructorID` as the PK:

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

By default, EF Core treats the key as non-database-generated because the column is for an identifying relationship. For more information, see EF Keys.

The Instructor navigation property

The `Instructor.OfficeAssignment` navigation property can be null because there might not be an `OfficeAssignment` row for a given instructor. An instructor might not have an office assignment.

The `OfficeAssignment.Instructor` navigation property will always have an instructor entity because the foreign key `InstructorID` type is `int`, a non-nullable value type. An office assignment can't exist without an instructor.

When an `Instructor` entity has a related `OfficeAssignment` entity, each entity has a reference to the other one in its navigation property.

The Course Entity

Update `Models/Course.cs` with the following code:

```
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<Instructor> Instructors { get; set; }
    }
}
```

The `Course` entity has a foreign key (FK) property `DepartmentID`. `DepartmentID` points to the related `Department` entity. The `Course` entity has a `Department` navigation property.

EF Core doesn't require a foreign key property for a data model when the model has a navigation property for a related entity. EF Core automatically creates FKs in the database wherever they're needed. EF Core creates shadow properties for automatically created FKs. However, explicitly including the FK in the data model can make updates simpler and more efficient. For example, consider a model where the FK property `DepartmentID` is *not* included. When a course entity is fetched to edit:

- The `Department` property is `null` if it's not explicitly loaded.
- To update the course entity, the `Department` entity must first be fetched.

When the FK property `DepartmentID` is included in the data model, there's no need to fetch the `Department` entity before an update.

The DatabaseGenerated attribute

The `[DatabaseGenerated(DatabaseGeneratedOption.None)]` attribute specifies that the PK is provided by the application rather than generated by the database.

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

By default, EF Core assumes that PK values are generated by the database. Database-generated is generally the best approach. For `Course` entities, the user specifies the PK. For example, a course number such as a 1000 series for the math department, a 2000 series for the English department.

The `DatabaseGenerated` attribute can also be used to generate default values. For example, the database can automatically generate a date field to record the date a row was created or updated. For more information, see Generated Properties.

Foreign key and navigation properties

The foreign key (FK) properties and navigation properties in the `Course` entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` FK and a `Department` navigation property.

```
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:

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

A course may be taught by multiple instructors, so the `Instructors` navigation property is a collection:

```
public ICollection<Instructor> Instructors { get; set; }
```

The Department entity

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

```
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; }

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

The Column attribute

Previously the `Column` attribute was used to change column name mapping. In the code for the `Department` entity, the `Column` attribute is used to change SQL data type mapping. The `Budget` column is defined using the SQL Server money type in the database:

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

Column mapping is generally not required. EF Core chooses the appropriate SQL Server data type based on the CLR type for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. `Budget` is for currency, and the money data type is more appropriate for currency.

Foreign key and navigation properties

The FK and navigation properties reflect the following relationships:

- A department may or may not have an administrator.
- An administrator is always an instructor. Therefore the `InstructorID` property is included as the FK to the `Instructor` entity.

The navigation property is named `Administrator` but holds an `Instructor` entity:

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

The `?` in the preceding code specifies the property is nullable.

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

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

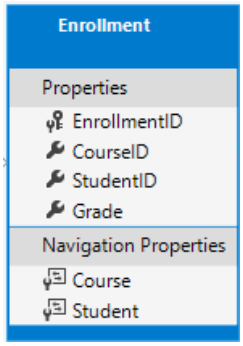
By convention, EF Core enables cascade delete for non-nullable FKs and for many-to-many relationships. This default behavior can result in circular cascade delete rules. Circular cascade delete rules cause an exception when a migration is added.

For example, if the `Department.InstructorID` property was defined as non-nullable, EF Core would configure a cascade delete rule. In that case, the department would be deleted when the instructor assigned as its administrator is deleted. In this scenario, a restrict rule would make more sense. The following fluent API would set a restrict rule and disable cascade delete.

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

The Enrollment foreign key and navigation properties

An enrollment record is for one course taken by one student.



Update `Models/Enrollment.cs` with the following code:

```
using System.ComponentModel.DataAnnotations;

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; }
    }
}
```

The FK properties and navigation properties reflect the following relationships:

An enrollment record is for one course, so there's a `CourseID` FK property and a `Course` navigation property:

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

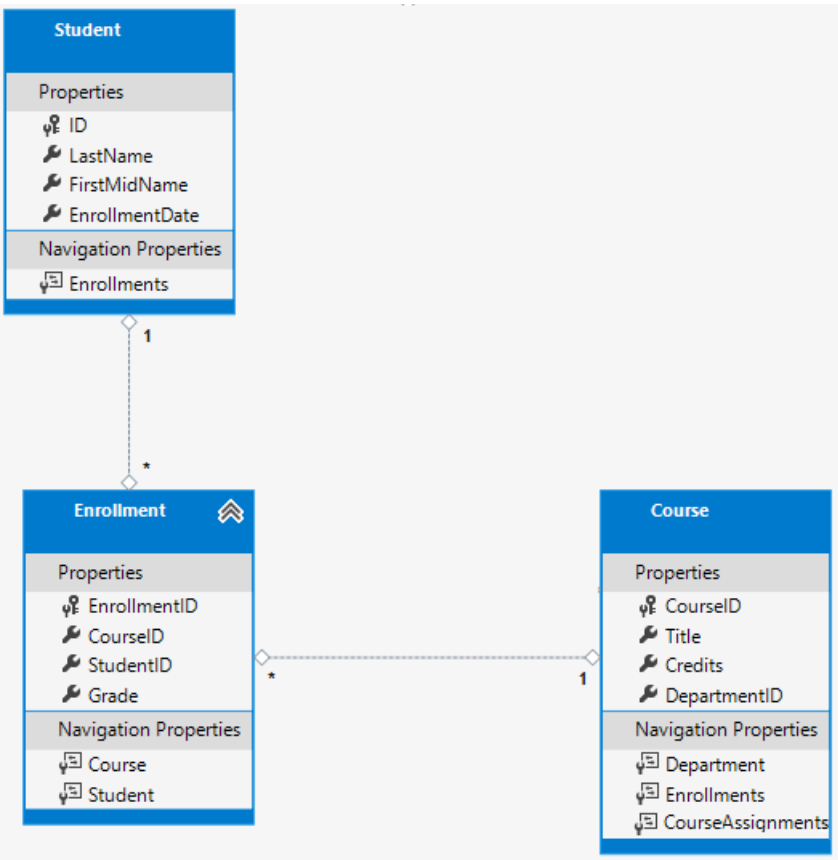
An enrollment record is for one student, so there's a `StudentID` FK property and a `Student` navigation property:

```
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. 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 FKs for the joined tables. In the `Enrollment` entity, the additional data besides FKs are the PK and `Grade`.

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using EF Power Tools for EF 6.x. Creating the diagram isn't part of the tutorial.)



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 FKs, `CourseID` and `StudentID`. A many-to-many join table without payload is sometimes called a pure join table (PJT).

The `Instructor` and `Course` entities have a many-to-many relationship using a PJT.

Update the database context

Update `Data/SchoolContext.cs` with the following code:

```
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; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable(nameof(Course))
                .HasMany(c => c.Instructors)
                .WithMany(i => i.Courses);
            modelBuilder.Entity<Student>().ToTable(nameof(Student));
            modelBuilder.Entity<Instructor>().ToTable(nameof(Instructor));
        }
    }
}
```

The preceding code adds the new entities and configures the many-to-many relationship between the `Instructor` and `Course` entities.

Fluent API alternative to attributes

The `OnModelCreating` method in the preceding code uses the *fluent API* to configure EF Core behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement. The following code is an example of the fluent API:

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

In this tutorial, the fluent API is used only for database mapping that can't be done with attributes. However, the fluent API can specify most of the formatting, validation, and mapping rules that can be done with attributes.

Some attributes such as `MinimumLength` can't be applied with the fluent API. `MinimumLength` doesn't change the schema, it only applies a minimum length validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes *clean*. Attributes and the fluent API can be mixed. There are some configurations that can only be done with the fluent API, for example, specifying a composite PK. There are some configurations that can only be done with attributes (`MinimumLength`). The recommended practice for using fluent API or attributes:

- Choose one of these two approaches.
- Use the chosen approach consistently as much as possible.

Some of the attributes used in this tutorial are used for:

- Validation only (for example, `MinimumLength`).
- EF Core configuration only (for example, `HasKey`).
- Validation and EF Core configuration (for example, `[StringLength(50)]`).

For more information about attributes vs. fluent API, see Methods of configuration.

Seed the database

Update the code in `Data/DbInitializer.cs`:

```
using ContosoUniversity.Models;
using System;
using System.Collections.Generic;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var alexander = new Student
            {
                FirstMidName = "Carson",
                LastName = "Alexander",
                EnrollmentDate = DateTime.Parse("2016-09-01")
            };

            var alonso = new Student
            {
                FirstMidName = "Meredith",
                LastName = "Alonso",
                EnrollmentDate = DateTime.Parse("2018-09-01")
            };

            var anand = new Student
            {
                FirstMidName = "Arturo",
                LastName = "Anand",
                EnrollmentDate = DateTime.Parse("2019-09-01")
            };

            var barzdukas = new Student
            {
                FirstMidName = "Gytis",
                LastName = "Barzdukas",
                EnrollmentDate = DateTime.Parse("2018-09-01")
            };

            var li = new Student
            {
                FirstMidName = "Yan",
                LastName = "Li",
                EnrollmentDate = DateTime.Parse("2018-09-01")
            };

            var justice = new Student
            {
                FirstMidName = "Peggy",
                LastName = "Justice",
```

```
            LastName = "Justice",
            EnrollmentDate = DateTime.Parse("2017-09-01")
        };

        var norman = new Student
        {
            FirstMidName = "Laura",
            LastName = "Norman",
            EnrollmentDate = DateTime.Parse("2019-09-01")
        };

        var olivetto = new Student
        {
            FirstMidName = "Nino",
            LastName = "Olivetto",
            EnrollmentDate = DateTime.Parse("2011-09-01")
        };

        var abercrombie = new Instructor
        {
            FirstMidName = "Kim",
            LastName = "Abercrombie",
            HireDate = DateTime.Parse("1995-03-11")
        };

        var fakhouri = new Instructor
        {
            FirstMidName = "Fadi",
            LastName = "Fakhouri",
            HireDate = DateTime.Parse("2002-07-06")
        };

        var harui = new Instructor
        {
            FirstMidName = "Roger",
            LastName = "Harui",
            HireDate = DateTime.Parse("1998-07-01")
        };

        var kapoor = new Instructor
        {
            FirstMidName = "Candace",
            LastName = "Kapoor",
            HireDate = DateTime.Parse("2001-01-15")
        };

        var zheng = new Instructor
        {
            FirstMidName = "Roger",
            LastName = "Zheng",
            HireDate = DateTime.Parse("2004-02-12")
        };

        var officeAssignments = new OfficeAssignment[]
        {
            new OfficeAssignment {
                Instructor = fakhouri,
                Location = "Smith 17" },
            new OfficeAssignment {
                Instructor = harui,
                Location = "Gowan 27" },
            new OfficeAssignment {
                Instructor = kapoor,
                Location = "Thompson 304" },
        };

        context.AddRange(officeAssignments);

        var english = new Department
```

```
            {
                Name = "English",
                Budget = 350000,
                StartDate = DateTime.Parse("2007-09-01"),
                Administrator = abercrombie
            };

            var mathematics = new Department
            {
                Name = "Mathematics",
                Budget = 100000,
                StartDate = DateTime.Parse("2007-09-01"),
                Administrator = fakhouri
            };

            var engineering = new Department
            {
                Name = "Engineering",
                Budget = 350000,
                StartDate = DateTime.Parse("2007-09-01"),
                Administrator = harui
            };

            var economics = new Department
            {
                Name = "Economics",
                Budget = 100000,
                StartDate = DateTime.Parse("2007-09-01"),
                Administrator = kapoor
            };

            var chemistry = new Course
            {
                CourseID = 1050,
                Title = "Chemistry",
                Credits = 3,
                Department = engineering,
                Instructors = new List<Instructor> { kapoor, harui }
            };

            var microeconomics = new Course
            {
                CourseID = 4022,
                Title = "Microeconomics",
                Credits = 3,
                Department = economics,
                Instructors = new List<Instructor> { zheng }
            };

            var macroeconmics = new Course
            {
                CourseID = 4041,
                Title = "Macroeconomics",
                Credits = 3,
                Department = economics,
                Instructors = new List<Instructor> { zheng }
            };

            var calculus = new Course
            {
                CourseID = 1045,
                Title = "Calculus",
                Credits = 4,
                Department = mathematics,
                Instructors = new List<Instructor> { fakhouri }
            };

            var trigonometry = new Course
            {
```

```
                CourseID = 3141,
                Title = "Trigonometry",
                Credits = 4,
                Department = mathematics,
                Instructors = new List<Instructor> { harui }
            };

            var composition = new Course
            {
                CourseID = 2021,
                Title = "Composition",
                Credits = 3,
                Department = english,
                Instructors = new List<Instructor> { abercrombie }
            };

            var literature = new Course
            {
                CourseID = 2042,
                Title = "Literature",
                Credits = 4,
                Department = english,
                Instructors = new List<Instructor> { abercrombie }
            };

            var enrollments = new Enrollment[]
            {
                new Enrollment {
                    Student = alexander,
                    Course = chemistry,
                    Grade = Grade.A
                },
                    new Enrollment {
                    Student = alexander,
                    Course = microeconomics,
                    Grade = Grade.C
                    },
                    new Enrollment {
                    Student = alexander,
                    Course = macroeconmics,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    Student = alonso,
                    Course = calculus,
                    Grade = Grade.B
                    },
                    new Enrollment {
                        Student = alonso,
                    Course = trigonometry,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    Student = alonso,
                    Course = composition,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    Student = anand,
                    Course = chemistry,
                    },
                    new Enrollment {
                    Student = anand,
                    Course = microeconomics,
                    Grade = Grade.B
                    },
                new Enrollment {
                    Student = barzdukas,
                    Course = chemistry,
```

```
                    Grade = Grade.B
                    },
                    new Enrollment {
                    Student = li,
                    Course = composition,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    Student = justice,
                    Course = literature,
                    Grade = Grade.B
                    }
            };

            context.AddRange(enrollments);
            context.SaveChanges();
        }
    }
}
```

The preceding code provides seed data for the new entities. Most of this code creates new entity objects and loads sample data. The sample data is used for testing.

Apply the migration or drop and re-create

With the existing database, there are two approaches to changing the database:

- Drop and re-create the database. Choose this section when using SQLite.
- Apply the migration to the existing database. The instructions in this section work for SQL Server only, ***not for SQLite***.

Either choice works for SQL Server. While the apply-migration method is more complex and time-consuming, it's the preferred approach for real-world, production environments.

Drop and re-create the database

To force EF Core to create a new database, drop and update the database:

- Visual Studio
- Visual Studio Code

- Delete the *Migrations* folder.
- In the **Package Manager Console** (PMC), run the following commands:

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

Run the app. Running the app runs the `DbInitializer.Initialize` method. The `DbInitializer.Initialize` populates the new database.

- Visual Studio
- Visual Studio Code

Open the database in SSOX:

- If SSOX was opened previously, click the **Refresh** button.

- Expand the **Tables** node. The created tables are displayed.

Next steps

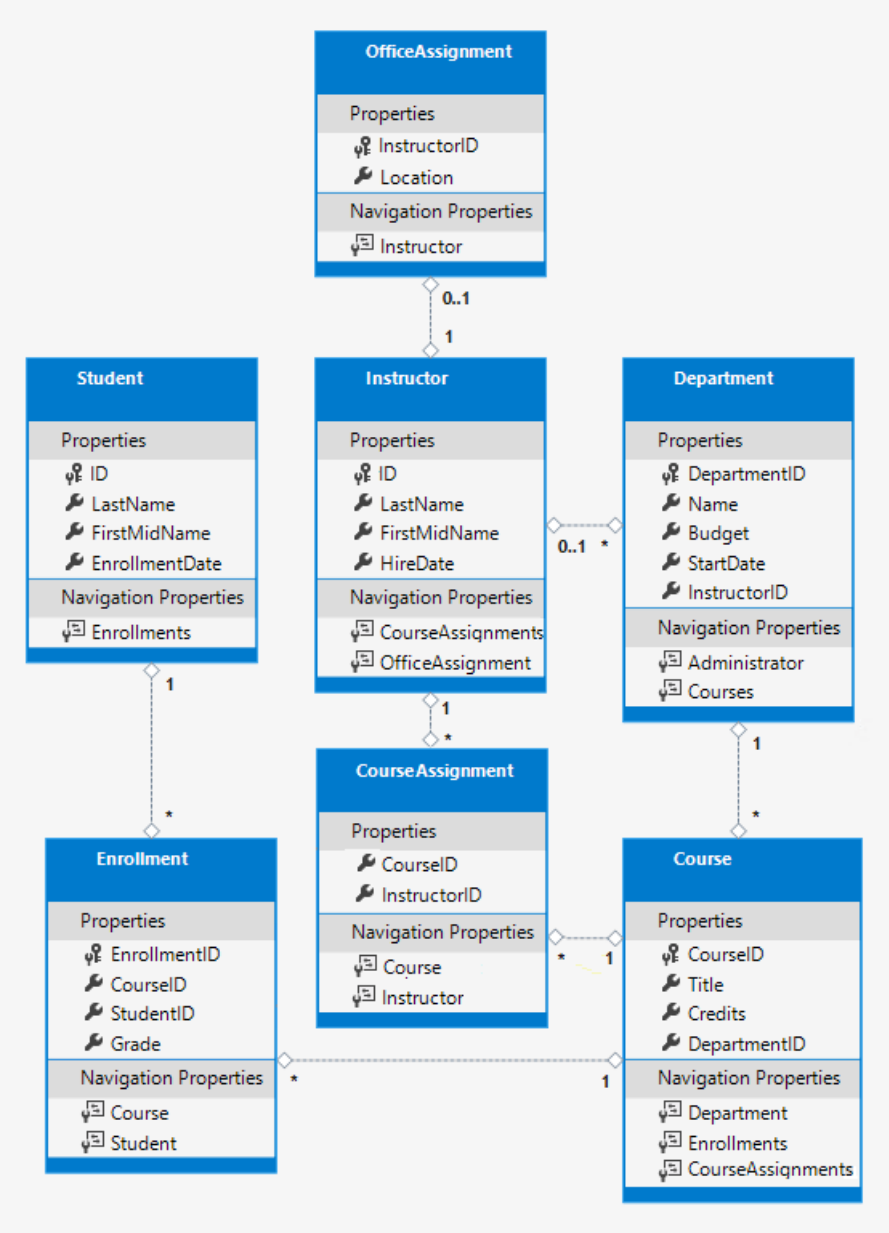
The next two tutorials show how to read and update related data.

PREVIOUS TUTORIAL | NEXT TUTORIAL

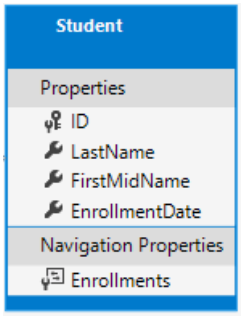
The previous tutorials worked with a basic data model that was composed of three entities. In this tutorial:

- More entities and relationships are added.
- The data model is customized by specifying formatting, validation, and database mapping rules.

The completed data model is shown in the following illustration:



The Student entity



Replace the code in `Models/Student.cs` with the following code:

```
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 preceding code adds a `FullName` property and adds the following attributes to existing properties:

- `[DataType]`
- `[DisplayFormat]`
- `[StringLength]`
- `[Column]`
- `[Required]`
- `[Display]`

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. `FullName` can't be set, so it has only a get accessor. No `FullName` column is created in the database.

The DataType attribute

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

For student enrollment dates, all of the pages currently display the time of day along with the date, although only the date is relevant. By using data annotation attributes, you can make one code change that will fix the display format in every page that shows the data.

The DataType attribute specifies a data type that's more specific than the database intrinsic type. In this case only the date should be displayed, not the date and time. The DataType Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, etc. The `DataType` attribute can also enable the app to automatically provide type-specific features. For example:

- The `mailto:` link is automatically created for `DataType.EmailAddress`.
- The date selector is provided for `DataType.Date` in most browsers.

The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes. The `DataType` attributes don't provide validation.

The DisplayFormat attribute

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

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the date 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. The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied to the edit UI. Some fields shouldn't use `ApplyFormatInEditMode`. For example, the currency symbol should generally not be displayed in an edit text box.

The `DisplayFormat` attribute can be used by itself. It's generally a good idea to use the `DataType` attribute with the `DisplayFormat` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen. The `DataType` attribute provides the following benefits that are not available in `DisplayFormat`:

- The browser can enable HTML5 features. For example, show a calendar control, the locale-appropriate currency symbol, email links, and client-side input validation.
- By default, the browser renders data using the correct format based on the locale.

For more information, see the <input> Tag Helper documentation.

The StringLength attribute

```
[StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
```

Data validation rules and validation error messages can be specified with attributes. The StringLength attribute specifies the minimum and maximum length of characters that are allowed in a data field. The code shown limits names to no more than 50 characters. An example that sets the minimum string length is shown later.

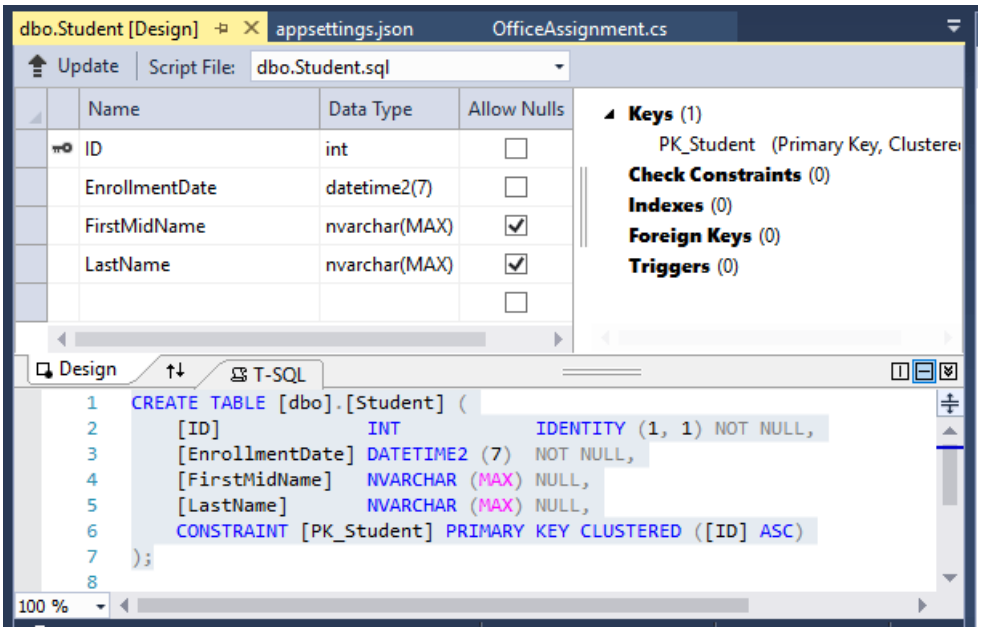
The `StringLength` attribute also provides client-side and server-side validation. The minimum value has no impact on the database schema.

The `StringLength` attribute doesn't prevent a user from entering white space for a name. The RegularExpression attribute can be used 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:

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

- Visual Studio
- Visual Studio Code

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



The preceding image shows the schema for the `Student` table. The name fields have type `nvarchar(MAX)`. When a migration is created and applied later in this tutorial, the name fields become `nvarchar(50)` as a result of the string length attributes.

The Column attribute

```
[Column("FirstName")]
public string FirstMidName { get; set; }
```

Attributes can control how classes and properties are mapped to the database. In the `Student` model, the `Column` attribute is used to map the name of the `FirstMidName` property to "FirstName" in the database.

When the database is created, property names on the model are used for column names (except when the `Column` attribute is used). The `Student` model uses `FirstMidName` for the first-name field because the field might also contain a middle name.

With the `[Column]` attribute, `Student.FirstMidName` in the data model maps to the `FirstName` column of the `Student` table. The addition of the `Column` attribute changes the model backing the `SchoolContext`. The model backing the `SchoolContext` no longer matches the database. That discrepancy will be resolved by adding a migration later in this tutorial.

The Required attribute

```
[Required]
```

The `Required` attribute makes the name properties required fields. The `Required` attribute isn't needed for non-nullable types such as value types (for example, `DateTime`, `int`, and `double`). Types that can't be null are automatically treated as required fields.

The `Required` attribute must be used with `MinimumLength` for the `MinimumLength` to be enforced.

```
[Display(Name = "Last Name")]
[Required]
[StringLength(50, MinimumLength=2)]
public string LastName { get; set; }
```

`MinimumLength` and `Required` allow whitespace to satisfy the validation. Use the `RegularExpression` attribute for full control over the string.

The Display attribute

```
[Display(Name = "Last Name")]
```

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date." The default captions had no space dividing the words, for example "Lastname."

Create a migration

Run the app and go to the Students page. An exception is thrown. The `[Column]` attribute causes EF to expect to find a column named `FirstName`, but the column name in the database is still `FirstMidName`.

- Visual Studio
- Visual Studio Code

The error message is similar to the following example:

```
SqlException: Invalid column name 'FirstName'.
```

- In the PMC, enter the following commands to create a new migration and update the database:

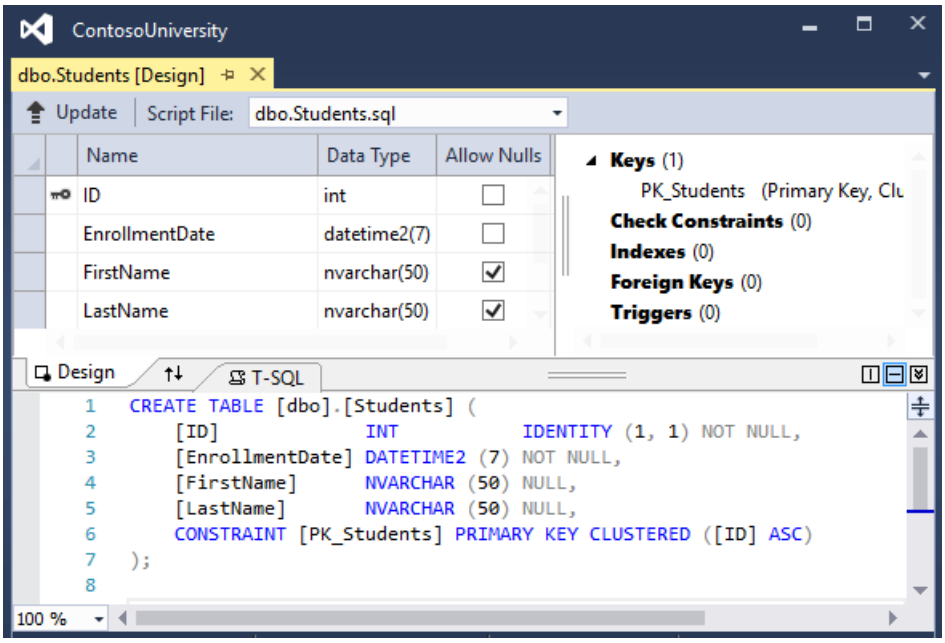
  ```
  Add-Migration ColumnFirstName
  Update-Database
  ```

  The first of these commands generates the following warning message:

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

  The warning is generated because the name fields are now limited to 50 characters. If a name in the database had more than 50 characters, the 51 to last character would be lost.

- Open the Student table in SSOX:

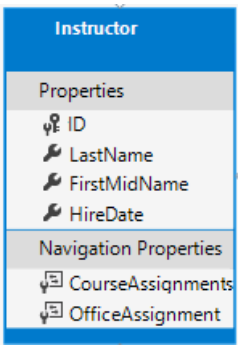


Before the migration was applied, the name columns were of type nvarchar(MAX). The name columns are now `nvarchar(50)`. The column name has changed from `FirstMidName` to `FirstName`.

- Run the app and go to the Students page.
- Notice that times are not input or displayed along with dates.
- Select **Create New**, and try to enter a name longer than 50 characters.

> **NOTE**
>
> In the following sections, building the app at some stages generates compiler errors. The instructions specify when to build the app.

The Instructor Entity



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

```
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; }
    }
}
```

Multiple attributes can be on one line. The `HireDate` attributes could be written as follows:

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

Navigation properties

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

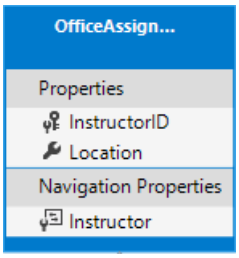
An instructor can teach any number of courses, so `CourseAssignments` is defined as a collection.

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

An instructor can have at most one office, so the `OfficeAssignment` property holds a single `OfficeAssignment` entity. `OfficeAssignment` is null if no office is assigned.

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

The OfficeAssignment entity



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

```
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

The `[Key]` attribute is used to identify a property as the primary key (PK) when the property name is something other than classnameID or ID.

There's a one-to-zero-or-one relationship between the `Instructor` and `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to. The `OfficeAssignment` PK is also its foreign key (FK) to the `Instructor` entity.

EF Core can't automatically recognize `InstructorID` as the PK of `OfficeAssignment` because `InstructorID` doesn't follow the ID or classnameID naming convention. Therefore, the `Key` attribute is used to identify `InstructorID` as the PK:

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

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

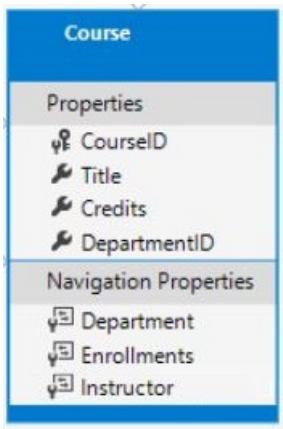
The Instructor navigation property

The `Instructor.OfficeAssignment` navigation property can be null because there might not be an `OfficeAssignment` row for a given instructor. An instructor might not have an office assignment.

The `OfficeAssignment.Instructor` navigation property will always have an instructor entity because the foreign key `InstructorID` type is `int`, a non-nullable value type. An office assignment can't exist without an instructor.

When an `Instructor` entity has a related `OfficeAssignment` entity, each entity has a reference to the other one in its navigation property.

The Course Entity



Update `Models/Course.cs` with the following code:

```
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 (FK) property `DepartmentID`. `DepartmentID` points to the related `Department` entity. The `Course` entity has a `Department` navigation property.

EF Core doesn't require a foreign key property for a data model when the model has a navigation property for a related entity. EF Core automatically creates FKs in the database wherever they're needed. EF Core creates shadow properties for automatically created FKs. However, explicitly including the FK in the data model can make updates simpler and more efficient. For example, consider a model where the FK property `DepartmentID` is *not* included. When a course entity is fetched to edit:

- The `Department` property is null if it's not explicitly loaded.
- To update the course entity, the `Department` entity must first be fetched.

When the FK property `DepartmentID` is included in the data model, there's no need to fetch the `Department` entity before an update.

The DatabaseGenerated attribute

The `[DatabaseGenerated(DatabaseGeneratedOption.None)]` attribute specifies that the PK is provided by the application rather than generated by the database.

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

By default, EF Core assumes that PK values are generated by the database. Database-generated is generally the best approach. For `Course` entities, the user specifies the PK. For example, a course number such as a 1000 series for the math department, a 2000 series for the English department.

The `DatabaseGenerated` attribute can also be used to generate default values. For example, the database can automatically generate a date field to record the date a row was created or updated. For more information, see Generated Properties.

Foreign key and navigation properties

The foreign key (FK) properties and navigation properties in the `Course` entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` FK and a `Department` navigation property.

```
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:

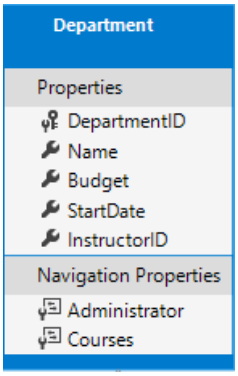
```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection:

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

`CourseAssignment` is explained later.

The Department entity



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

```
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; }

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

The Column attribute

Previously the `Column` attribute was used to change column name mapping. In the code for the `Department` entity, the `Column` attribute is used to change SQL data type mapping. The `Budget` column is defined using the SQL Server money type in the database:

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

Column mapping is generally not required. EF Core chooses the appropriate SQL Server data type based on the CLR type for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. `Budget` is for currency, and the money data type is more appropriate for currency.

Foreign key and navigation properties

The FK and navigation properties reflect the following relationships:

- A department may or may not have an administrator.
- An administrator is always an instructor. Therefore the `InstructorID` property is included as the FK to the `Instructor` entity.

The navigation property is named `Administrator` but holds an `Instructor` entity:

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

The question mark (?) in the preceding code specifies the property is nullable.

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

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

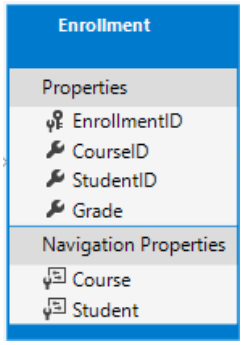
By convention, EF Core enables cascade delete for non-nullable FKs and for many-to-many relationships. This default behavior can result in circular cascade delete rules. Circular cascade delete rules cause an exception when a migration is added.

For example, if the `Department.InstructorID` property was defined as non-nullable, EF Core would configure a cascade delete rule. In that case, the department would be deleted when the instructor assigned as its administrator is deleted. In this scenario, a restrict rule would make more sense. The following fluent API would set a restrict rule and disable cascade delete.

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

The Enrollment entity

An enrollment record is for one course taken by one student.



Update `Models/Enrollment.cs` with the following code:

```
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 FK properties and navigation properties reflect the following relationships:

An enrollment record is for one course, so there's a `CourseID` FK property and a `Course` navigation property:

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

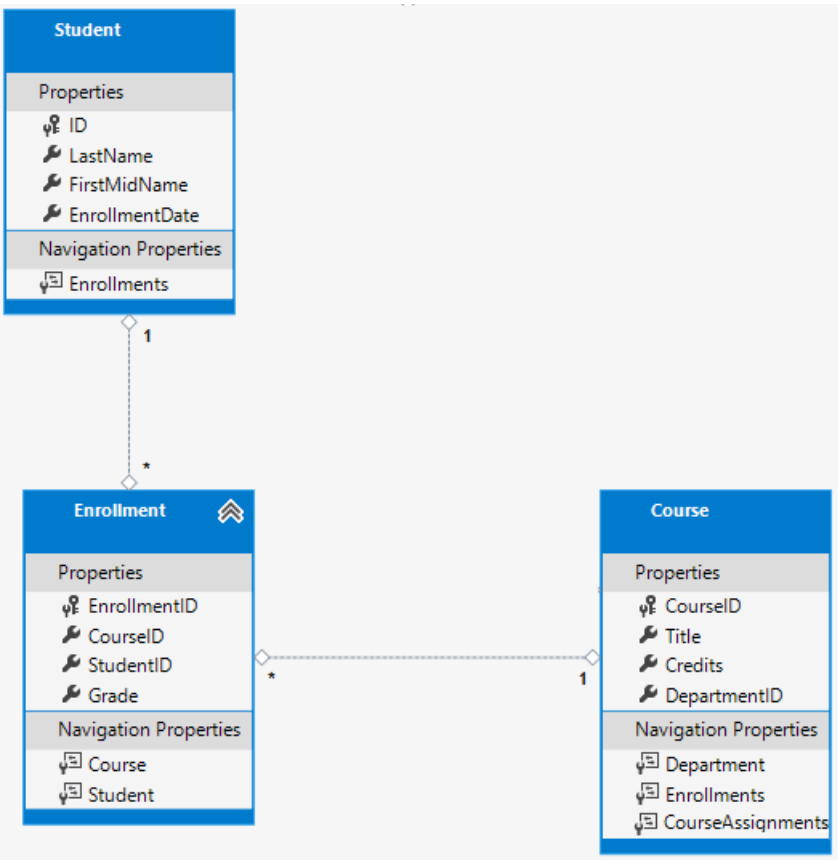
An enrollment record is for one student, so there's a `StudentID` FK property and a `Student` navigation property:

```
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. 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 FKs for the joined tables (in this case, the PK and `Grade`).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using EF Power Tools for EF 6.x. Creating the diagram isn't part of the tutorial.)



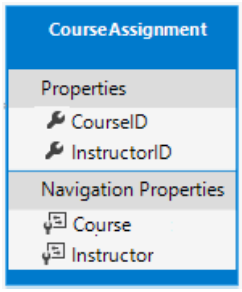
Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the `Enrollment` table didn't include grade information, it would only need to contain the two FKs (`CourseID` and `StudentID`). A many-to-many join table without payload is sometimes called a pure join table (PJT).

The `Instructor` and `Course` entities have a many-to-many relationship using a pure join table.

Note: EF 6.x supports implicit join tables for many-to-many relationships, but EF Core doesn't. For more information, see Many-to-many relationships in EF Core 2.0.

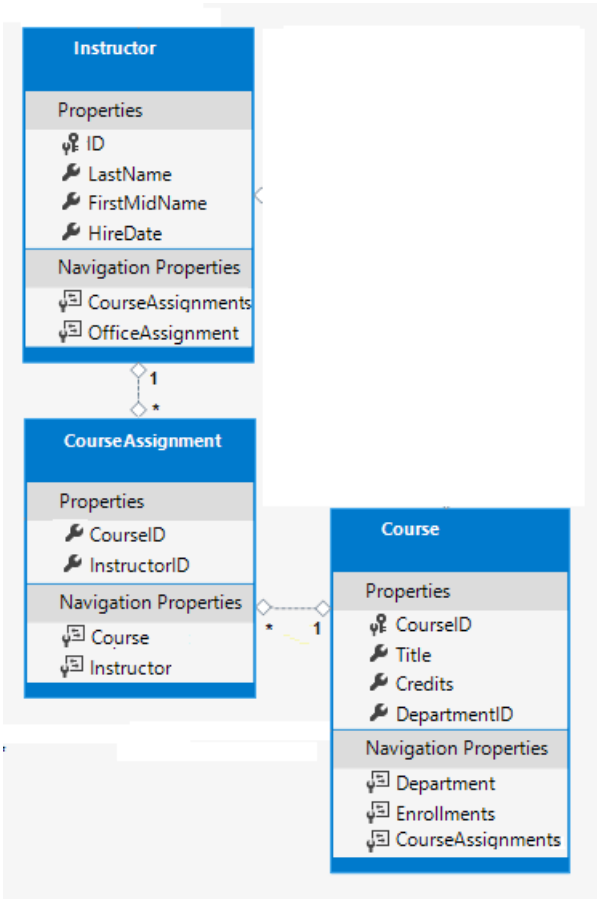
The CourseAssignment entity



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

```
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; }
    }
}
```

The Instructor-to-Courses many-to-many relationship requires a join table, and the entity for that join table is CourseAssignment.



It's common to name a join entity `EntityName1EntityName2`. For example, the Instructor-to-Courses join table using this pattern would be `CourseInstructor`. However, we recommend using a name that describes the relationship.

Data models start out simple and grow. Join tables without payload (PJTs) frequently evolve to include payload. By starting with a descriptive entity name, the name doesn't need to change when the join table changes. 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 with a join entity called Ratings. For the Instructor-to-Courses many-to-many relationship, `CourseAssignment` is preferred over `CourseInstructor`.

Composite key

The two FKs in `CourseAssignment` (`InstructorID` and `CourseID`) together uniquely identify each row of the `CourseAssignment` table. `CourseAssignment` doesn't require a dedicated PK. The `InstructorID` and `CourseID` properties function as a composite PK. The only way to specify composite PKs to EF Core is with the *fluent API*. The next section shows how to configure the composite PK.

The composite key ensures that:

- Multiple rows are allowed for one course.
- Multiple rows are allowed for one instructor.
- Multiple rows aren't allowed for the same instructor and course.

The `Enrollment` join entity defines its own PK, so duplicates of this sort are possible. To prevent such duplicates:

- Add a unique index on the FK fields, or
- Configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see Indexes.

Update the database context

Update `Data/SchoolContext.cs` with the following code:

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

The preceding code adds the new entities and configures the `CourseAssignment` entity's composite PK.

Fluent API alternative to attributes

The `OnModelCreating` method in the preceding code uses the *fluent API* to configure EF Core behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement. The following code is an example of the fluent API:

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

In this tutorial, the fluent API is used only for database mapping that can't be done with attributes. However, the fluent API can specify most of the formatting, validation, and mapping rules that can be done with attributes.

Some attributes such as `MinimumLength` can't be applied with the fluent API. `MinimumLength` doesn't change the schema, it only applies a minimum length validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." Attributes and the fluent API can be mixed. There are some configurations that can only be done with the fluent API (specifying a composite PK). There are some configurations that can only be done with attributes (`MinimumLength`). The recommended practice for using fluent API or attributes:

- Choose one of these two approaches.
- Use the chosen approach consistently as much as possible.

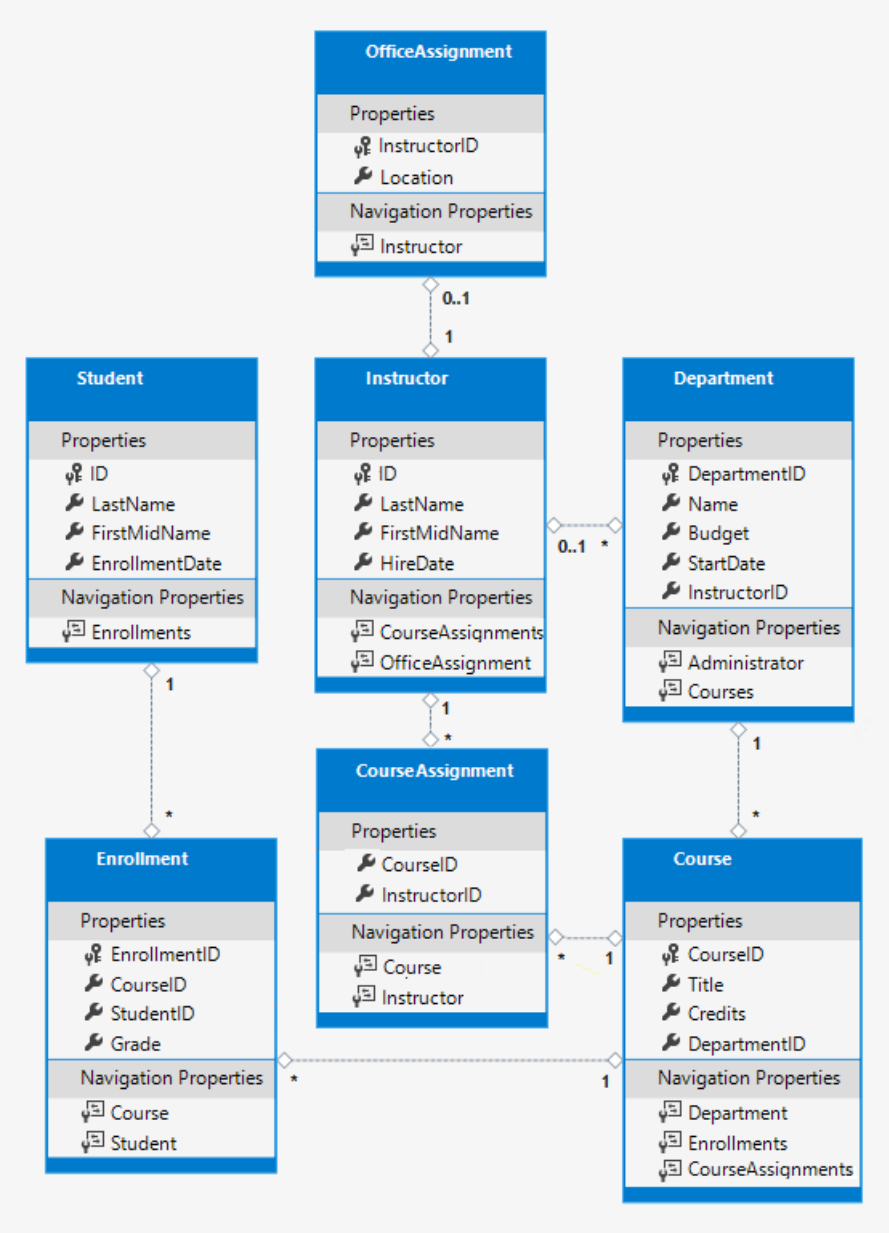
Some of the attributes used in this tutorial are used for:

- Validation only (for example, `MinimumLength`).
- EF Core configuration only (for example, `HasKey`).
- Validation and EF Core configuration (for example, `[StringLength(50)]`).

For more information about attributes vs. fluent API, see Methods of configuration.

Entity diagram

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



The preceding diagram shows:

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

Seed the database

Update the code in `Data/DbInitializer.cs`:

```
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("2016-09-01") },
                new Student { FirstMidName = "Meredith", LastName = "Alonso",
                    EnrollmentDate = DateTime.Parse("2018-09-01") },
                new Student { FirstMidName = "Arturo",   LastName = "Anand",
                    EnrollmentDate = DateTime.Parse("2019-09-01") },
                new Student { FirstMidName = "Gytis",    LastName = "Barzdukas",
                    EnrollmentDate = DateTime.Parse("2018-09-01") },
                new Student { FirstMidName = "Yan",      LastName = "Li",
                    EnrollmentDate = DateTime.Parse("2018-09-01") },
                new Student { FirstMidName = "Peggy",    LastName = "Justice",
                    EnrollmentDate = DateTime.Parse("2017-09-01") },
                new Student { FirstMidName = "Laura",    LastName = "Norman",
                    EnrollmentDate = DateTime.Parse("2019-09-01") },
                new Student { FirstMidName = "Nino",     LastName = "Olivetto",
                    EnrollmentDate = DateTime.Parse("2011-09-01") }
            };

            context.Students.AddRange(students);
            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") }
            };

            context.Instructors.AddRange(instructors);
            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 }
            };

            context.Departments.AddRange(departments);
            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
        },
    };

    context.Courses.AddRange(courses);
    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" },
    };

    context.OfficeAssignments.AddRange(officeAssignments);
    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
```

```
            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
            },
    };

    context.CourseAssignments.AddRange(courseInstructors);
    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)
```

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

The preceding code provides seed data for the new entities. Most of this code creates new entity objects and loads sample data. The sample data is used for testing. See `Enrollments` and `CourseAssignments` for examples of how many-to-many join tables can be seeded.

Add a migration

Build the project.

- Visual Studio
- Visual Studio Code

In PMC, run the following command.

```
Add-Migration ComplexDataModel
```

The preceding command displays a warning about possible data loss.

```
An operation was scaffolded that may result in the loss of data.
Please review the migration for accuracy.
To undo this action, use 'ef migrations remove'
```

If the `database update` command is run, the following error is produced:

```
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'.
```

In the next section, you see what to do about this error.

Apply the migration or drop and re-create

Now that you have an existing database, you need to think about how to apply changes to it. This tutorial shows two alternatives:

- Drop and re-create the database. Choose this section if you're using SQLite.
- Apply the migration to the existing database. The instructions in this section work for SQL Server only, **not for SQLite**.

Either choice works for SQL Server. While the apply-migration method is more complex and time-consuming, it's the preferred approach for real-world, production environments.

Drop and re-create the database

Skip this section if you're using SQL Server and want to do the apply-migration approach in the following section.

To force EF Core to create a new database, drop and update the database:

- Visual Studio
- Visual Studio Code

- In the **Package Manager Console** (PMC), run the following command:

```
Drop-Database
```

- Delete the *Migrations* folder, then run the following command:

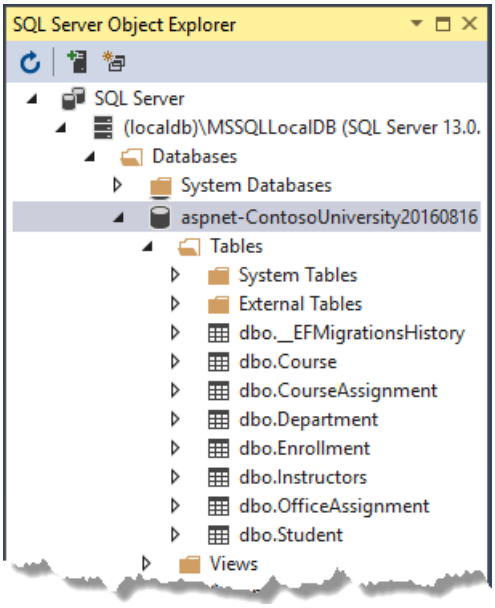
```
Add-Migration InitialCreate
Update-Database
```

Run the app. Running the app runs the `DbInitializer.Initialize` method. The `DbInitializer.Initialize` populates the new database.

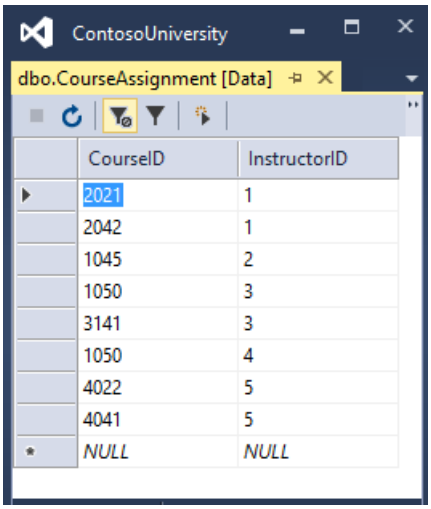
- Visual Studio
- Visual Studio Code

Open the database in SSOX:

- If SSOX was opened previously, click the **Refresh** button.
- Expand the **Tables** node. The created tables are displayed.



- Examine the **CourseAssignment** table:
  - Right-click the **CourseAssignment** table and select **View Data**.
  - Verify the **CourseAssignment** table contains data.



Apply the migration

This section is optional. These steps work only for SQL Server LocalDB and only if you skipped the preceding Drop and re-create the database section.

When migrations are run with existing data, there may be FK constraints that are not satisfied with the existing data. With production data, steps must be taken to migrate the existing data. This section provides an example of fixing FK constraint violations. Don't make these code changes without a backup. Don't make these code changes if you completed the preceding Drop and re-create the database section.

The `{timestamp}_ComplexDataModel.cs` file contains the following code:

```
migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    type: "int",
    nullable: false,
    defaultValue: 0);
```

The preceding code adds a non-nullable `DepartmentID` FK to the `Course` table. The database from the previous tutorial contains rows in `Course`, so that table cannot be updated by migrations.

To make the `ComplexDataModel` migration work with existing data:

- Change the code to give the new column (`DepartmentID`) a default value.
- Create a fake department named "Temp" to act as the default department.

Fix the foreign key constraints

In the `ComplexDataModel` migration class, update the `Up` method:

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

```
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. The new code goes after the `.CreateTable( name: "Department"` block:

```
migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(type: "int", nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(type: "int", nullable: true),
        Name = table.Column<string>(type: "nvarchar(50)", maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(type: "datetime2", 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);
```

With the preceding changes, existing `Course` rows will be related to the "Temp" department after the `ComplexDataModel.Up` method runs.

The way of handling the situation shown here is simplified for this tutorial. A production app would:

- Include code or scripts to add `Department` rows and related `Course` rows to the new `Department` rows.
- Not use the "Temp" department or the default value for `Course.DepartmentID`.

- Visual Studio
- Visual Studio Code

- In the **Package Manager Console** (PMC), run the following command:

```
Update-Database
```

Because the `DbInitializer.Initialize` method is designed to work only with an empty database, use SSOX to delete all the rows in the Student and Course tables. (Cascade delete will take care of the Enrollment table.)

Run the app. Running the app runs the `DbInitializer.Initialize` method. The `DbInitializer.Initialize` populates the new database.

Next steps

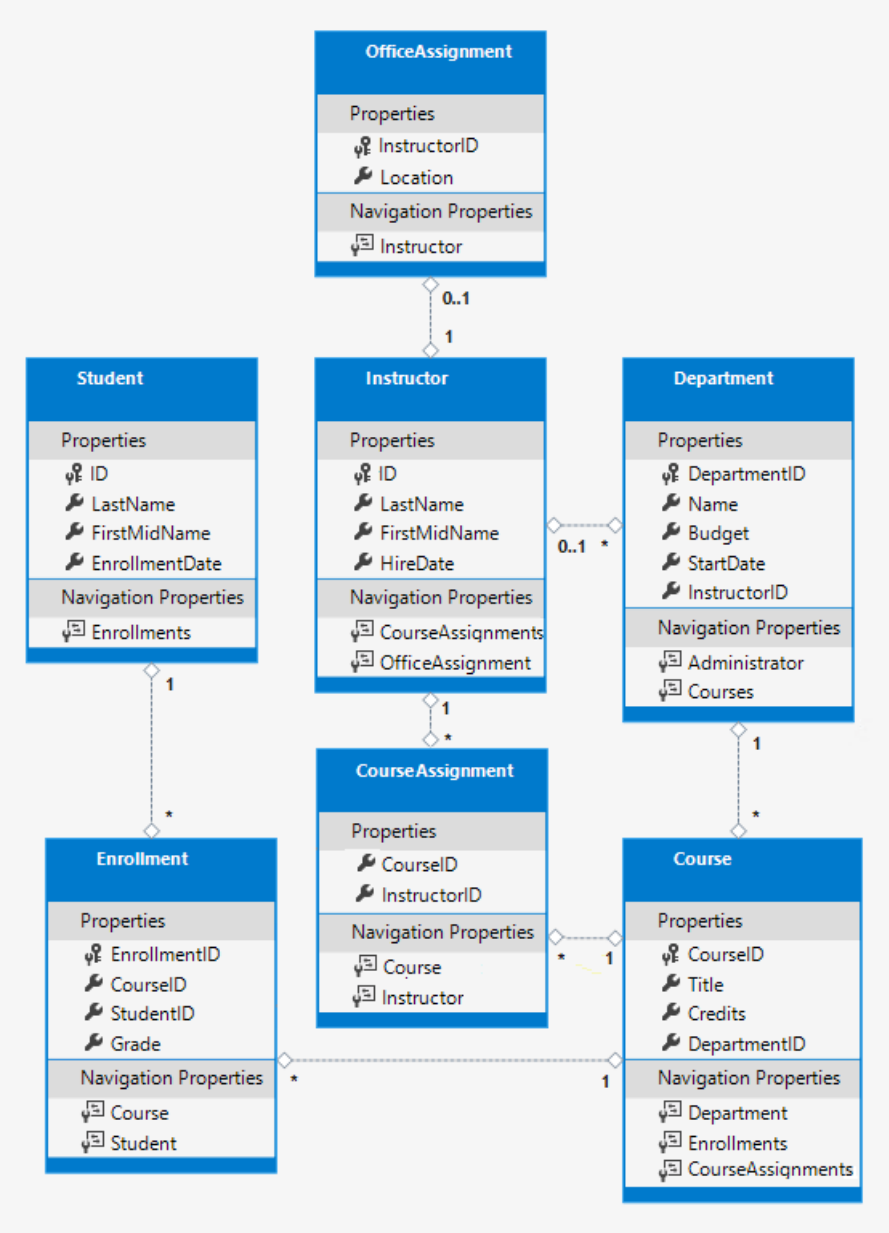
The next two tutorials show how to read and update related data.

PREVIOUS TUTORIAL | NEXT TUTORIAL

The previous tutorials worked with a basic data model that was composed of three entities. In this tutorial:

- More entities and relationships are added.
- The data model is customized by specifying formatting, validation, and database mapping rules.

The entity classes for the completed data model are shown in the following illustration:



If you run into problems you can't solve, download the completed app.

Customize the data model with attributes

In this section, the data model is customized using attributes.

The DataType attribute

The student pages currently displays the time of the enrollment date. Typically, date fields show only the date and not the time.

Update `Models/Student.cs` with the following highlighted code:

```
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 specifies a data type that's more specific than the database intrinsic type. In this case only the date should be displayed, not the date and time. The DataType Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, etc. The `DataType` attribute can also enable the app to automatically provide type-specific features. For example:

- The `mailto:` link is automatically created for `DataType.EmailAddress`.
- The date selector is provided for `DataType.Date` in most browsers.

The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers consume. The `DataType` attributes don't provide validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the date 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:

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

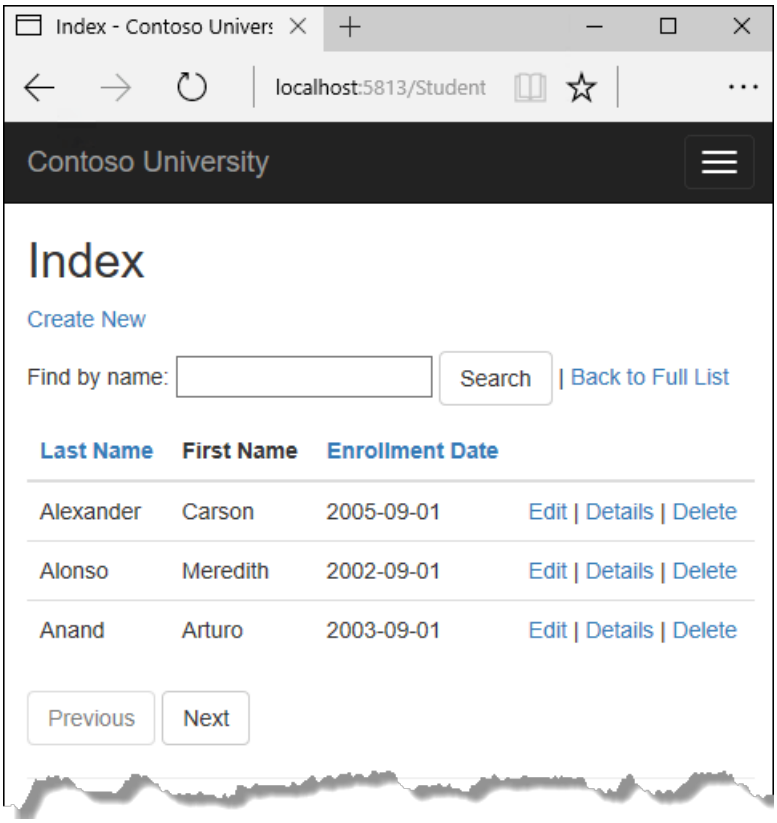
The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied to the edit UI. Some fields shouldn't use `ApplyFormatInEditMode`. For example, the currency symbol should generally not be displayed in an edit text box.

The `DisplayFormat` attribute can be used by itself. It's generally a good idea to use the `DataType` attribute with the `DisplayFormat` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen. The `DataType` attribute provides the following benefits that are not available in `DisplayFormat`:

- The browser can enable HTML5 features. For example, show a calendar control, the locale-appropriate currency symbol, email links, client-side input validation, etc.
- By default, the browser renders data using the correct format based on the locale.

For more information, see the <input> Tag Helper documentation.

Run the app. Navigate to the Students Index page. Times are no longer displayed. Every view that uses the `Student` model displays the date without time.



The StringLength attribute

Data validation rules and validation error messages can be specified with attributes. The StringLength attribute specifies the minimum and maximum length of characters that are allowed in a data field. The `StringLength` attribute also provides client-side and server-side validation. The minimum value has no impact on the database schema.

Update the `Student` model with the following code:

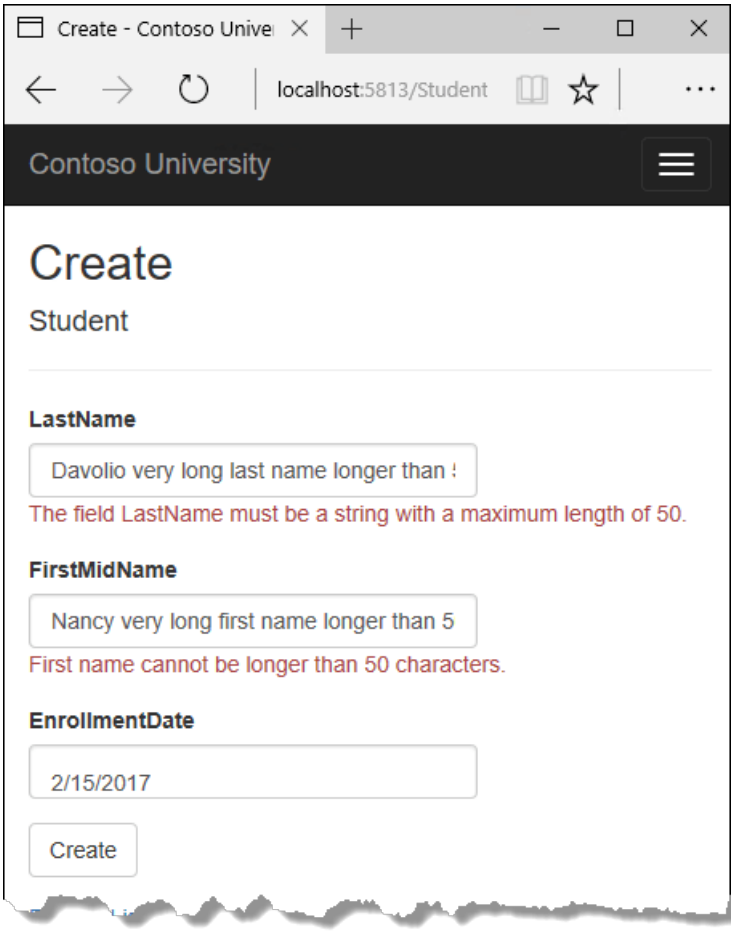
```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

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 preceding code limits names to no more than 50 characters. The `StringLength` attribute doesn't prevent a user from entering white space for a name. The RegularExpression attribute is used 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:

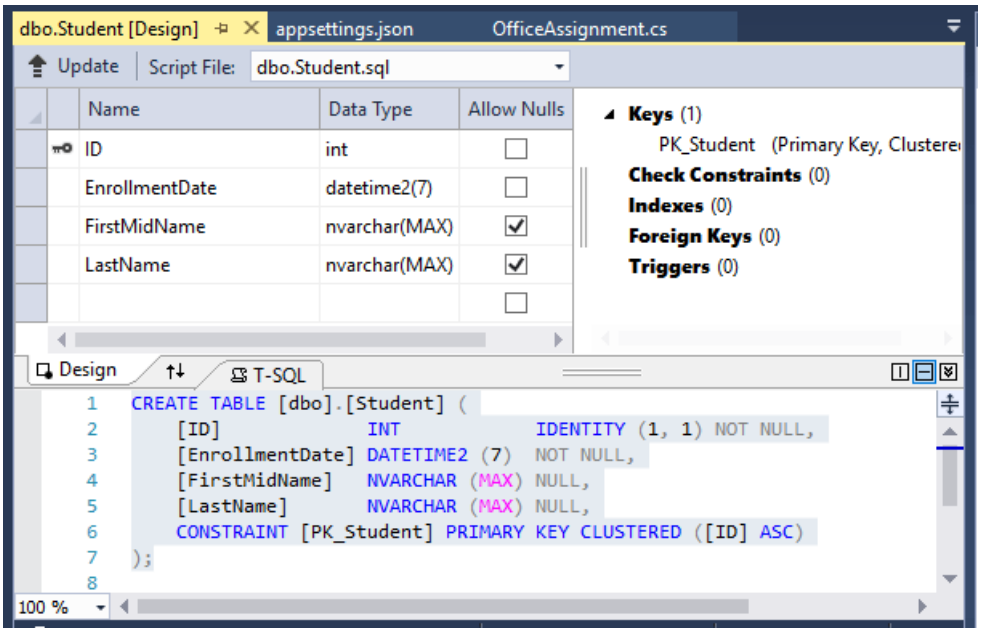
```
[RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
```

Run the app:

- Navigate to the Students page.
- Select **Create New**, and enter a name longer than 50 characters.
- Select **Create**, client-side validation shows an error message.



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



The preceding image shows the schema for the `Student` table. The name fields have type `nvarchar(MAX)` because migrations has not been run on the DB. When migrations are run later in this tutorial, the name fields

become `nvarchar(50)`.

The Column attribute

Attributes can control how classes and properties are mapped to the database. In this section, the `Column` attribute is used to map the name of the `FirstMidName` property to "FirstName" in the DB.

When the DB is created, property names on the model are used for column names (except when the `Column` attribute is used).

The `Student` model uses `FirstMidName` for the first-name field because the field might also contain a middle name.

Update the `Student.cs` file with the following highlighted code:

```
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; }
    }
}
```

With the preceding change, `Student.FirstMidName` in the app maps to the `FirstName` column of the `Student` table.

The addition of the `Column` attribute changes the model backing the `SchoolContext`. The model backing the `SchoolContext` no longer matches the database. If the app is run before applying migrations, the following exception is generated:

```
SqlException: Invalid column name 'FirstName'.
```

To update the DB:

- Build the project.
- Open a command window in the project folder. Enter the following commands to create a new migration and update the DB:

- Visual Studio
- Visual Studio Code

```
Add-Migration ColumnFirstName
Update-Database
```

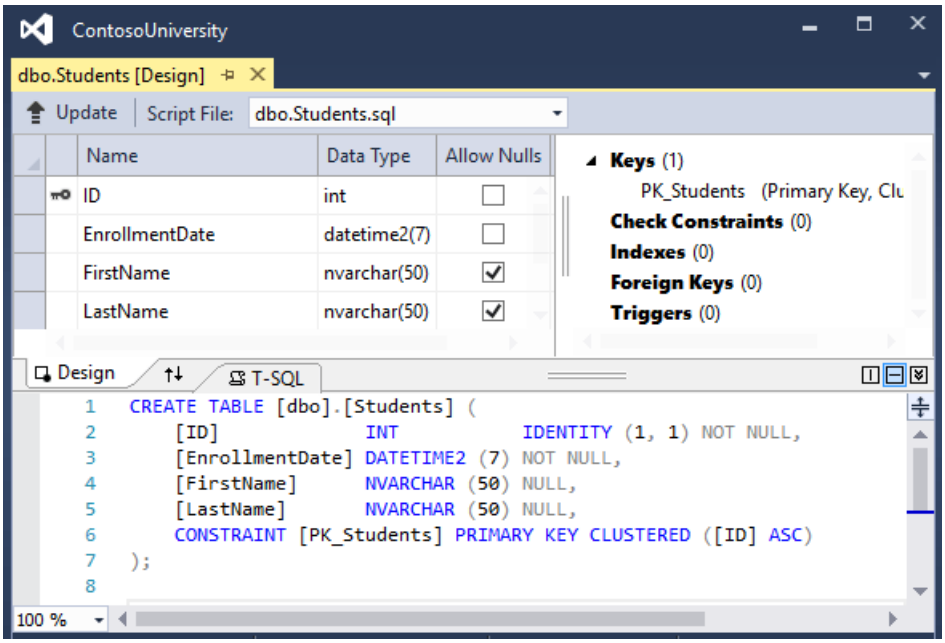
The `migrations add ColumnFirstName` command generates the following warning message:

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

The warning is generated because the name fields are now limited to 50 characters. If a name in the DB had more than 50 characters, the 51 to last character would be lost.

- Test the app.

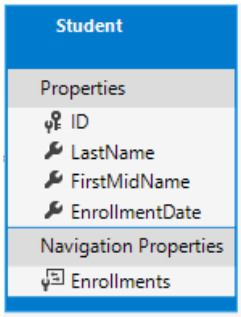
Open the Student table in SSOX:



Before migration was applied, the name columns were of type nvarchar(MAX). The name columns are now `nvarchar(50)`. The column name has changed from `FirstMidName` to `FirstName`.

> **NOTE**
> In the following section, building the app at some stages generates compiler errors. The instructions specify when to build the app.

Student entity update



Update `Models/Student.cs` with the following code:

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

The `Required` attribute could be replaced with a minimum length parameter in the `StringLength` attribute:

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

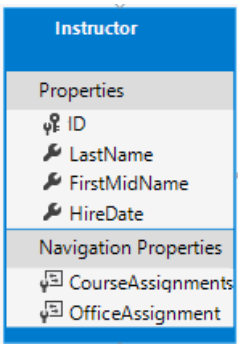
The Display attribute

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date." The default captions had no space dividing the words, for example "Lastname."

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. `FullName` cannot be set, it has only a get accessor. No `FullName` column is created in the database.

Create the Instructor Entity



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

```
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; }
    }
}
```

Multiple attributes can be on one line. The `HireDate` attributes could be written as follows:

```
[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.

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

If a navigation property holds multiple entities:

- It must be a list type where the entries can be added, deleted, and updated.

Navigation property types include:

- `ICollection<T>`
- `List<T>`
- `HashSet<T>`

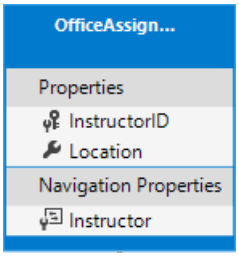
If `ICollection<T>` is specified, EF Core creates a `HashSet<T>` collection by default.

The `CourseAssignment` entity is explained in the section on many-to-many relationships.

Contoso University business rules state that an instructor can have at most one office. The `OfficeAssignment` property holds a single `OfficeAssignment` entity. `OfficeAssignment` is null if no office is assigned.

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

Create the OfficeAssignment entity



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

```
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

The `[Key]` attribute is used to identify a property as the primary key (PK) when the property name is something other than classnameID or ID.

There's a one-to-zero-or-one relationship between the `Instructor` and `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to. The `OfficeAssignment` PK is also its foreign

key (FK) to the `Instructor` entity. EF Core can't automatically recognize `InstructorID` as the PK of `OfficeAssignment` because:

- `InstructorID` doesn't follow the ID or classnameID naming convention.

Therefore, the `Key` attribute is used to identify `InstructorID` as the PK:

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

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

The Instructor navigation property

The `OfficeAssignment` navigation property for the `Instructor` entity is nullable because:

- Reference types (such as classes are nullable).
- An instructor might not have an office assignment.

The `OfficeAssignment` entity has a non-nullable `Instructor` navigation property because:

- `InstructorID` is non-nullable.
- An office assignment can't exist without an instructor.

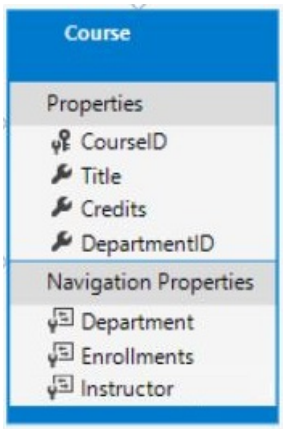
When an `Instructor` entity has a related `OfficeAssignment` entity, each entity has a reference to the other one in its navigation property.

The `[Required]` attribute could be applied to the `Instructor` navigation property:

```
[Required]
public Instructor Instructor { get; set; }
```

The preceding code specifies that there must be a related instructor. The preceding code is unnecessary because the `InstructorID` foreign key (which is also the PK) is non-nullable.

Modify the Course Entity



Update `Models/Course.cs` with the following code:

```
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 (FK) property `DepartmentID`. `DepartmentID` points to the related `Department` entity. The `Course` entity has a `Department` navigation property.

EF Core doesn't require a FK property for a data model when the model has a navigation property for a related entity.

EF Core automatically creates FKs in the database wherever they're needed. EF Core creates shadow properties for automatically created FKs. Having the FK in the data model can make updates simpler and more efficient. For example, consider a model where the FK property `DepartmentID` is *not* included. When a course entity is fetched to edit:

- The `Department` entity is null if it's not explicitly loaded.
- To update the course entity, the `Department` entity must first be fetched.

When the FK property `DepartmentID` is included in the data model, there's no need to fetch the `Department` entity before an update.

The DatabaseGenerated attribute

The `[DatabaseGenerated(DatabaseGeneratedOption.None)]` attribute specifies that the PK is provided by the application rather than generated by the database.

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

By default, EF Core assumes that PK values are generated by the DB. DB generated PK values is generally the best approach. For `Course` entities, the user specifies the PK. For example, a course number such as a 1000 series for the math department, a 2000 series for the English department.

The `DatabaseGenerated` attribute can also be used to generate default values. For example, the DB can automatically generate a date field to record the date a row was created or updated. For more information, see Generated Properties.

Foreign key and navigation properties

The foreign key (FK) properties and navigation properties in the `Course` entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` FK and a `Department` navigation property.

```
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:

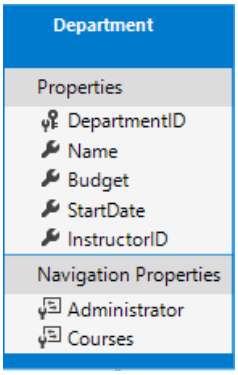
```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection:

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

`CourseAssignment` is explained later.

Create the Department entity



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

```
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; }

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

The Column attribute

Previously the `Column` attribute was used to change column name mapping. In the code for the `Department` entity, the `Column` attribute is used to change SQL data type mapping. The `Budget` column is defined using the SQL Server money type in the DB:

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

Column mapping is generally not required. EF Core generally chooses the appropriate SQL Server data type based on the CLR type for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. `Budget` is for currency, and the money data type is more appropriate for currency.

Foreign key and navigation properties

The FK and navigation properties reflect the following relationships:

- A department may or may not have an administrator.
- An administrator is always an instructor. Therefore the `InstructorID` property is included as the FK to the `Instructor` entity.

The navigation property is named `Administrator` but holds an `Instructor` entity:

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

The question mark (?) in the preceding code specifies the property is nullable.

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

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

Note: By convention, EF Core enables cascade delete for non-nullable FKs and for many-to-many relationships. Cascading delete can result in circular cascade delete rules. Circular cascade delete rules causes an exception when a migration is added.

For example, if the `Department.InstructorID` property was defined as non-nullable:

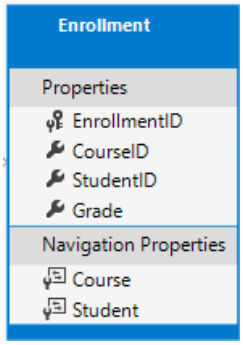
- EF Core configures a cascade delete rule to delete the department when the instructor is deleted.
- Deleting the department when the instructor is deleted isn't the intended behavior.
- The following fluent API would set a restrict rule instead of cascade.

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

The preceding code disables cascade delete on the department-instructor relationship.

Update the Enrollment entity

An enrollment record is for one course taken by one student.



Update `Models/Enrollment.cs` with the following code:

```
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 FK properties and navigation properties reflect the following relationships:

An enrollment record is for one course, so there's a `CourseID` FK property and a `Course` navigation property:

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

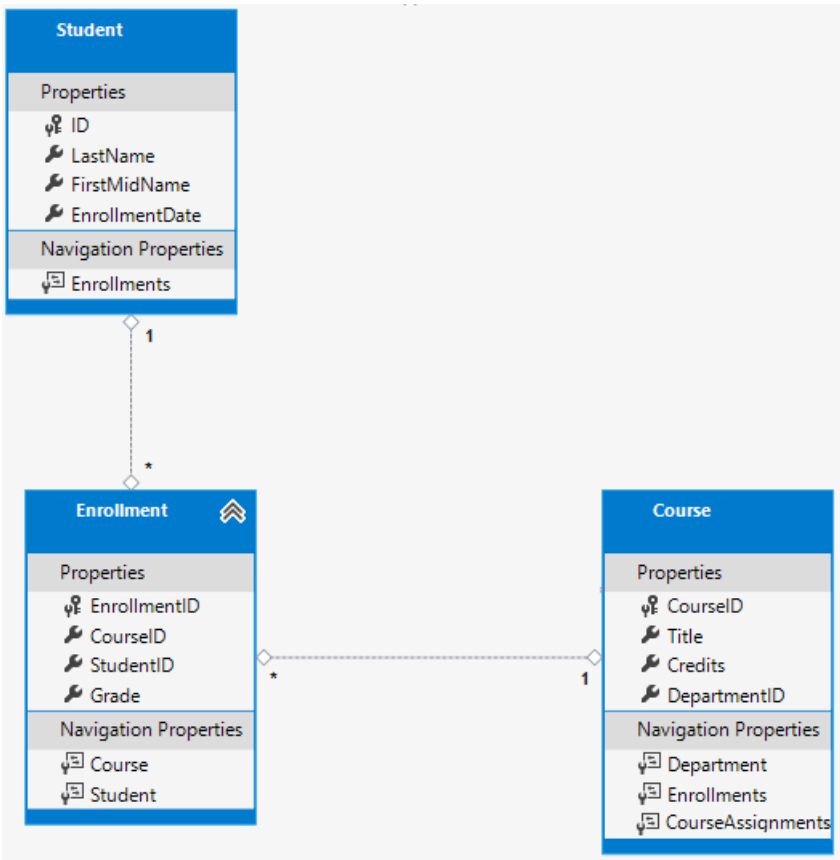
An enrollment record is for one student, so there's a `StudentID` FK property and a `Student` navigation property:

```
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. 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 FKs for the joined tables (in this case, the PK and `Grade`).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using EF Power Tools for EF 6.x. Creating the diagram isn't part of the tutorial.)



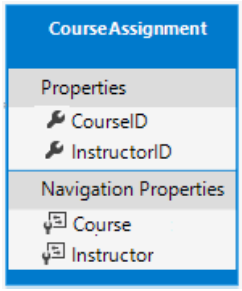
Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the `Enrollment` table didn't include grade information, it would only need to contain the two FKs (`CourseID` and `StudentID`). A many-to-many join table without payload is sometimes called a pure join table (PJT).

The `Instructor` and `Course` entities have a many-to-many relationship using a pure join table.

Note: EF 6.x supports implicit join tables for many-to-many relationships, but EF Core doesn't. For more information, see Many-to-many relationships in EF Core 2.0.

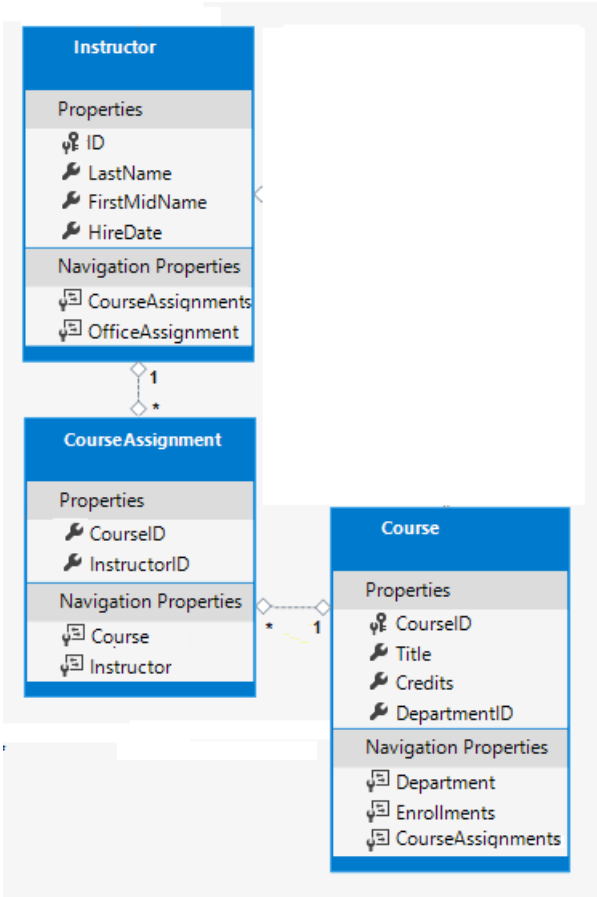
The CourseAssignment entity



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

```
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; }
    }
}
```

Instructor-to-Courses



The Instructor-to-Courses many-to-many relationship:

- Requires a join table that must be represented by an entity set.
- Is a pure join table (table without payload).

It's common to name a join entity `EntityName1EntityName2`. For example, the Instructor-to-Courses join table using this pattern is `CourseInstructor`. However, we recommend using a name that describes the relationship.

Data models start out simple and grow. No-payload joins (PJTs) frequently evolve to include payload. By starting with a descriptive entity name, the name doesn't need to change when the join table changes. 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 with a join entity called Ratings. For the Instructor-to-Courses many-to-many relationship, `CourseAssignment` is preferred over `CourseInstructor`.

Composite key

FKs are not nullable. The two FKs in `CourseAssignment` (`InstructorID` and `CourseID`) together uniquely identify each row of the `CourseAssignment` table. `CourseAssignment` doesn't require a dedicated PK. The `InstructorID` and `CourseID` properties function as a composite PK. The only way to specify composite PKs to EF Core is with the *fluent API*. The next section shows how to configure the composite PK.

The composite key ensures:

- Multiple rows are allowed for one course.
- Multiple rows are allowed for one instructor.
- Multiple rows for the same instructor and course isn't allowed.

The `Enrollment` join entity defines its own PK, so duplicates of this sort are possible. To prevent such duplicates:

- Add a unique index on the FK fields, or
- Configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see Indexes.

Update the DB context

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

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

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

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

The preceding code adds the new entities and configures the `CourseAssignment` entity's composite PK.

Fluent API alternative to attributes

The `OnModelCreating` method in the preceding code uses the *fluent API* to configure EF Core behavior. The API is

called "fluent" because it's often used by stringing a series of method calls together into a single statement. The following code is an example of the fluent API:

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

In this tutorial, the fluent API is used only for DB mapping that can't be done with attributes. However, the fluent API can specify most of the formatting, validation, and mapping rules that can be done with attributes.

Some attributes such as `MinimumLength` can't be applied with the fluent API. `MinimumLength` doesn't change the schema, it only applies a minimum length validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." Attributes and the fluent API can be mixed. There are some configurations that can only be done with the fluent API (specifying a composite PK). There are some configurations that can only be done with attributes (`MinimumLength`). The recommended practice for using fluent API or attributes:

- Choose one of these two approaches.
- Use the chosen approach consistently as much as possible.

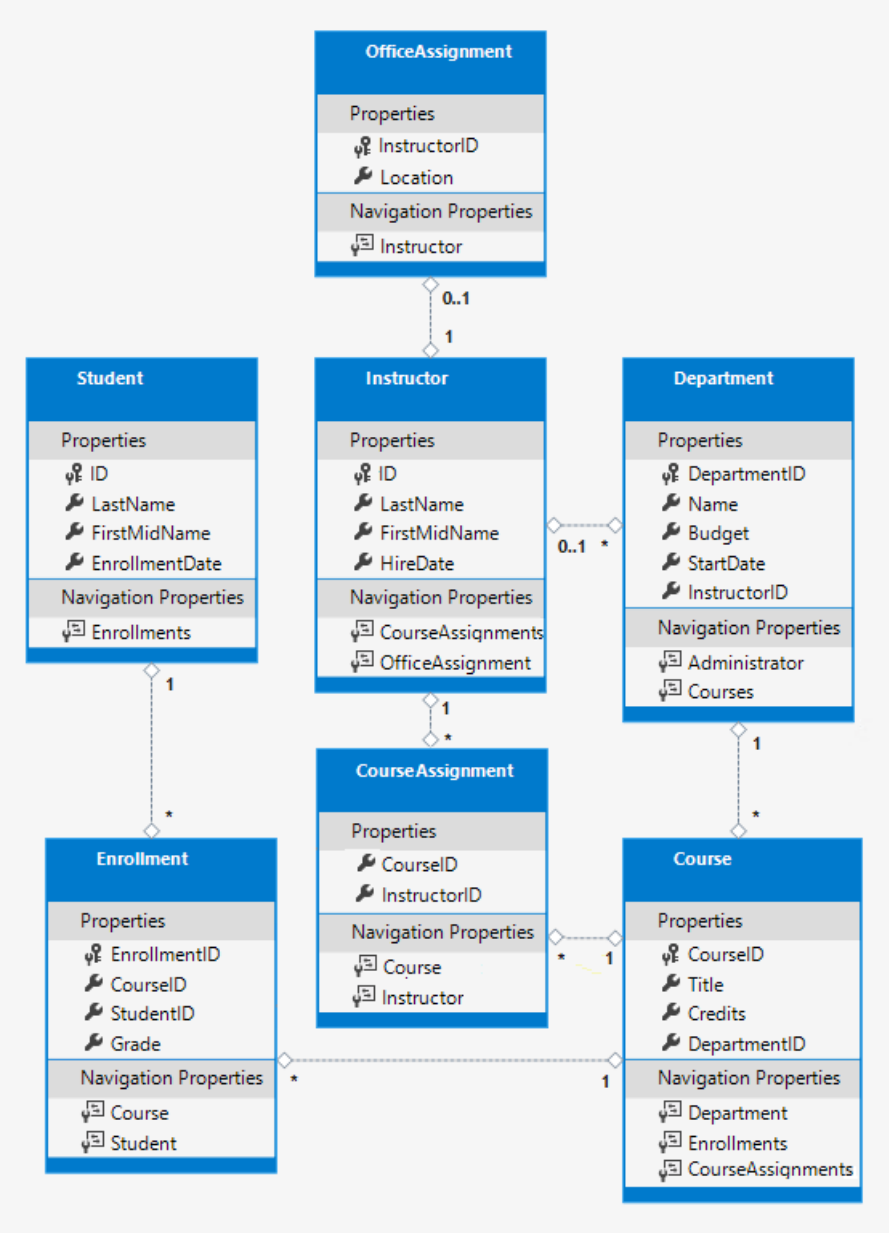
Some of the attributes used in the this tutorial are used for:

- Validation only (for example, `MinimumLength`).
- EF Core configuration only (for example, `HasKey`).
- Validation and EF Core configuration (for example, `[StringLength(50)]`).

For more information about attributes vs. fluent API, see Methods of configuration.

Entity Diagram Showing Relationships

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



The preceding diagram shows:

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

Seed the DB with Test Data

Update the code in `Data/DbInitializer.cs`:

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

The preceding code provides seed data for the new entities. Most of this code creates new entity objects and loads sample data. The sample data is used for testing. See `Enrollments` and `CourseAssignments` for examples of how many-to-many join tables can be seeded.

Add a migration

Build the project.

- Visual Studio
- Visual Studio Code

```
Add-Migration ComplexDataModel
```

The preceding command displays a warning about possible data loss.

```
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 the `database update` command is run, the following error is produced:

```
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'.
```

Apply the migration

Now that you have an existing database, you need to think about how to apply future changes to it. This tutorial shows two approaches:

- Drop and re-create the database
- Apply the migration to the existing database. While this method is more complex and time-consuming, it's the preferred approach for real-world, production environments. **Note**: This is an optional section of the tutorial. You can do the drop and re-create steps and skip this section. If you do want to follow the steps in this section, don't do the drop and re-create steps.

Drop and re-create the database

The code in the updated `DbInitializer` adds seed data for the new entities. To force EF Core to create a new DB, drop and update the DB:

- Visual Studio
- Visual Studio Code

In the **Package Manager Console** (PMC), run the following command:

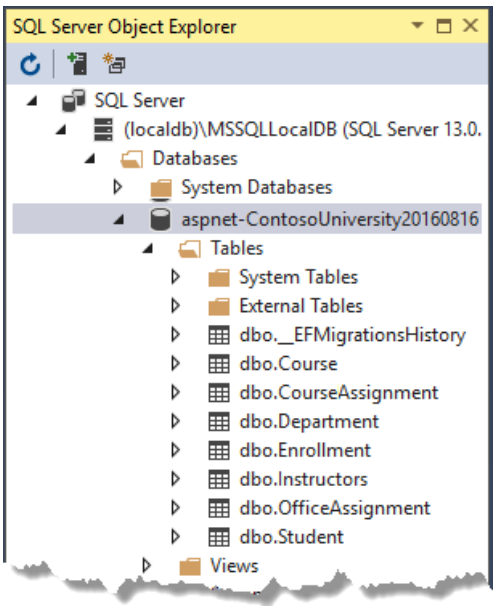
```
Drop-Database
Update-Database
```

Run `Get-Help about_EntityFrameworkCore` from the PMC to get help information.

Run the app. Running the app runs the `DbInitializer.Initialize` method. The `DbInitializer.Initialize` populates the new DB.

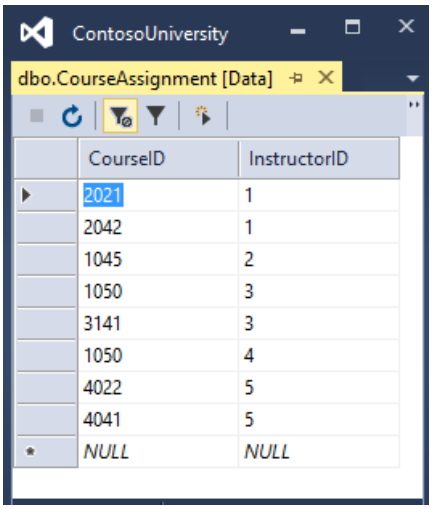
Open the DB in SSOX:

- If SSOX was opened previously, click the **Refresh** button.
- Expand the **Tables** node. The created tables are displayed.



Examine the **CourseAssignment** table:

- Right-click the **CourseAssignment** table and select **View Data**.
- Verify the **CourseAssignment** table contains data.



Apply the migration to the existing database

This section is optional. These steps work only if you skipped the preceding Drop and re-create the database section.

When migrations are run with existing data, there may be FK constraints that are not satisfied with the existing data. With production data, steps must be taken to migrate the existing data. This section provides an example of fixing FK constraint violations. Don't make these code changes without a backup. Don't make these code changes if you completed the previous section and updated the database.

The `{timestamp}_ComplexDataModel.cs` file contains the following code:

```
migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    type: "int",
    nullable: false,
    defaultValue: 0);
```

The preceding code adds a non-nullable `DepartmentID` FK to the `Course` table. The DB from the previous tutorial contains rows in `Course`, so that table cannot be updated by migrations.

To make the `ComplexDataModel` migration work with existing data:

- Change the code to give the new column (`DepartmentID`) a default value.
- Create a fake department named "Temp" to act as the default department.

Fix the foreign key constraints

Update the `ComplexDataModel` classes `Up` method:

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

```
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. The new code goes after the `.CreateTable( name: "Department"` block:

```
migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(type: "int", nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(type: "int", nullable: true),
        Name = table.Column<string>(type: "nvarchar(50)", maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(type: "datetime2", 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);
```

With the preceding changes, existing `Course` rows will be related to the "Temp" department after the `ComplexDataModel` `Up` method runs.

A production app would:

- Include code or scripts to add `Department` rows and related `Course` rows to the new `Department` rows.
- Not use the "Temp" department or the default value for `Course.DepartmentID`.

The next tutorial covers related data.

Additional resources

- YouTube version of this tutorial(Part 1)
- YouTube version of this tutorial(Part 2)

PREVIOUS NEXT

Part 6, Razor Pages with EF Core in ASP.NET Core - Read Related Data

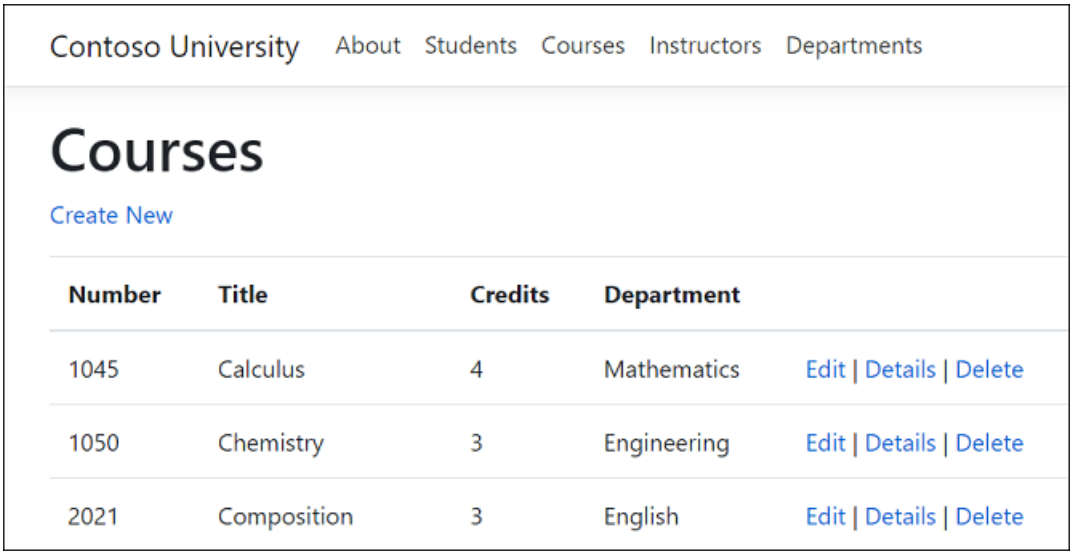
By Tom Dykstra, Jon P Smith, and Rick Anderson

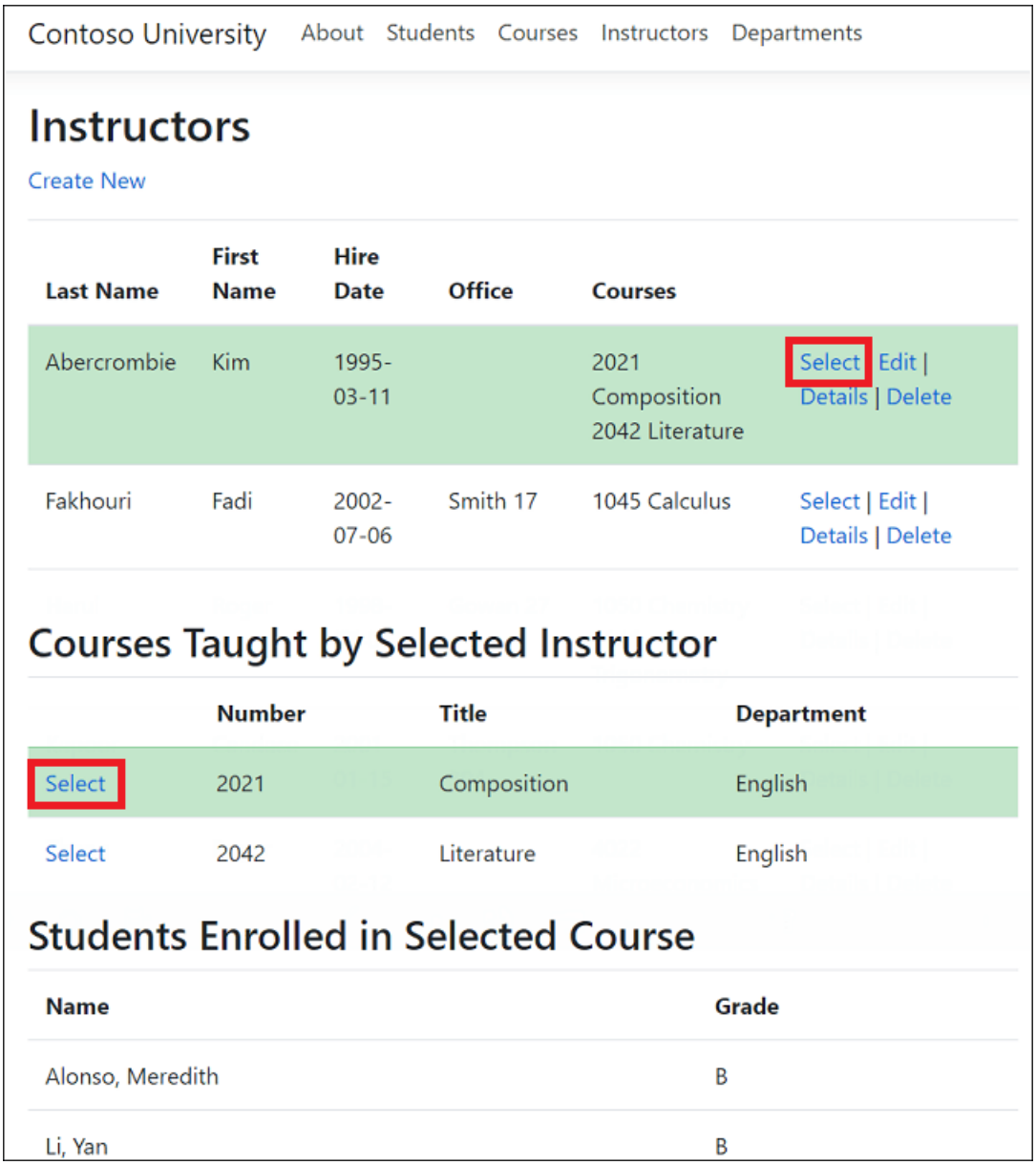
The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial shows how to read and display related data. Related data is data that EF Core loads into navigation properties.

The following illustrations show the completed pages for this tutorial:





Eager, explicit, and lazy loading

There are several ways that EF Core can load related data into the navigation properties of an entity:

- Eager loading. Eager loading is when a query for one type of entity also loads related entities. When an entity is read, its related data is retrieved. This typically results in a single join query that retrieves all of the data that's needed. EF Core will issue multiple queries for some types of eager loading. Issuing multiple queries can be more efficient than a large single query. Eager loading is specified with 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

  Eager loading sends multiple queries when a collection navigation is included:

  - One query for the main query
  - One query for each collection "edge" in the load tree.

- Separate queries with `Load`: The data can be retrieved in separate queries, and EF Core "fixes up" the navigation properties. "Fixes up" means that EF Core automatically populates the navigation properties.

Separate queries with `Load` is more like explicit loading than eager loading.

```
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

Note: EF Core automatically fixes up navigation properties to any other entities that were previously loaded into the context instance. Even if the data for a navigation property is *not* explicitly included, the property may still be populated if some or all of the related entities were previously loaded.

- Explicit loading. When the entity is first read, related data isn't retrieved. Code must be written to retrieve the related data when it's needed. Explicit loading with separate queries results in multiple queries sent to the database. With explicit loading, the code specifies the navigation properties to be loaded. 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);
    }
}
```

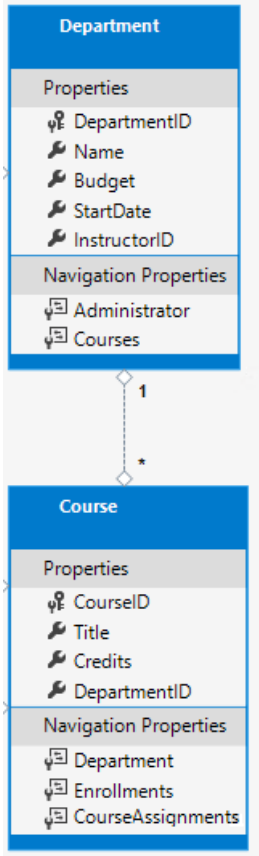
Query: all Department rows

Query: Course rows related to Department d

- Lazy loading. When the entity is first read, related data isn't retrieved. The first time a navigation property is accessed, the data required for that navigation property is automatically retrieved. A query is sent to the database each time a navigation property is accessed for the first time. Lazy loading can hurt performance, for example when developers use N+1 queries. N+1 queries load a parent and enumerate through children.

Create Course pages

The `Course` entity includes a navigation property that contains the related `Department` entity.



To display the name of the assigned department for a course:

- Load the related `Department` entity into the `Course.Department` navigation property.
- Get the name from the `Department` entity's `Name` property.

Scaffold Course pages

- Visual Studio
- Visual Studio Code

- Follow the instructions in Scaffold Student pages with the following exceptions:
  - Create a *Pages/Courses* folder.
  - Use `Course` for the model class.
  - Use the existing context class instead of creating a new one.
- Open `Pages/Courses/Index.cshtml.cs` and examine the `OnGetAsync` method. The scaffolding engine specified eager loading for the `Department` navigation property. The `Include` method specifies eager loading.
- Run the app and select the **Courses** link. The department column displays the `DepartmentID`, which isn't useful.

Display the department name

Update Pages/Courses/Index.cshtml.cs with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IList<Course> Courses { get; set; }

        public async Task OnGetAsync()
        {
            Courses = await _context.Courses
                .Include(c => c.Department)
                .AsNoTracking()
                .ToListAsync();
        }
    }
}
```

The preceding code changes the `Course` property to `Courses` and adds `AsNoTracking`. `AsNoTracking` improves performance because the entities returned are not tracked. The entities don't need to be tracked because they're not updated in the current context.

Update `Pages/Courses/Index.cshtml` with the following code.

```
@page
@model ContosoUniversity.Pages.Courses.IndexModel

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

<h1>Courses</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Courses)
{
        <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-page="./Edit" asp-route-id="@item.CourseID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.CourseID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.CourseID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

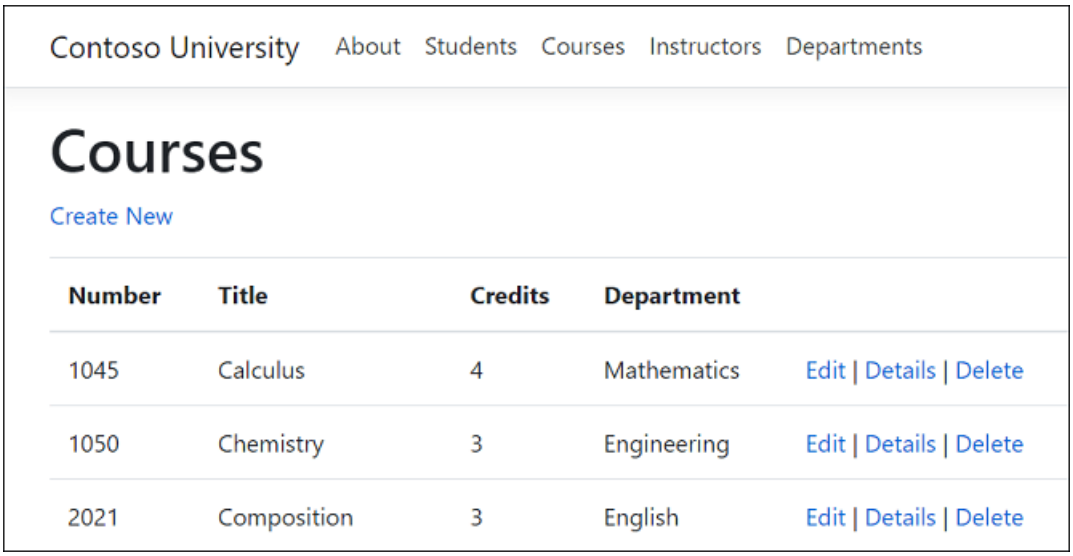
The following changes have been made to the scaffolded code:

- Changed the `Course` property name to `Courses`.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they're meaningless to end users. However, in this case the primary key is meaningful.
- 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:

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

Run the app and select the **Courses** tab to see the list with department names.



Loading related data with Select

The `OnGetAsync` method loads related data with the `Include` method. The `Select` method is an alternative that loads only the related data needed. For single items, like the `Department.Name` it uses a `SQL INNER JOIN`. For collections, it uses another database access, but so does the `Include` operator on collections.

The following code loads related data with the `Select` method:

```
public IList<CourseViewModel> CourseVM { get; set; }

public async Task OnGetAsync()
{
    CourseVM = await _context.Courses
    .Select(p => new CourseViewModel
    {
        CourseID = p.CourseID,
        Title = p.Title,
        Credits = p.Credits,
        DepartmentName = p.Department.Name
    }).ToListAsync();
}
```

The preceding code doesn't return any entity types, therefore no tracking is done. For more information about the EF tracking, see Tracking vs. No-Tracking Queries.

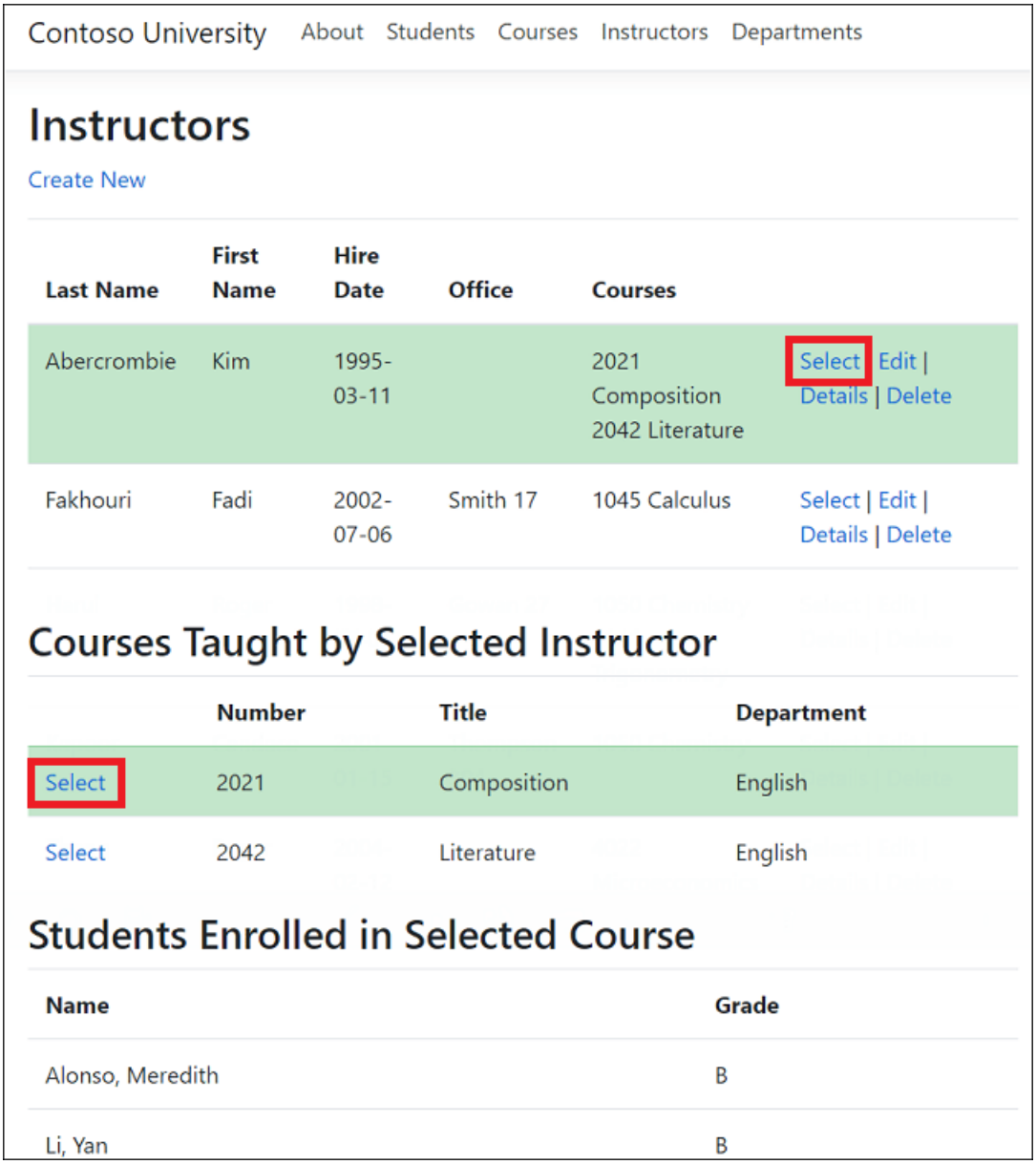
The `CourseViewModel`:

```
public class CourseViewModel
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public string DepartmentName { get; set; }
}
```

See IndexSelectModel for the complete Razor Pages.

Create Instructor pages

This section scaffolds Instructor pages and adds related Courses and Enrollments to the Instructors Index page.



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

- The list of instructors displays related data from the `OfficeAssignment` entity (Office in the preceding image). The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. Eager loading is used for the `OfficeAssignment` entities. Eager loading is typically more efficient when the related data needs to be displayed. In this case, office assignments for the instructors are displayed.
- When the user selects an instructor, related `Course` entities are displayed. The `Instructor` and `Course` entities are in a many-to-many relationship. Eager loading is used for the `Course` entities and their related `Department` entities. In this case, separate queries might be more efficient because only courses for the selected instructor are needed. This example shows how to use eager loading for navigation properties in entities that are in navigation properties.
- When the user selects a course, related data from the `Enrollments` entity is displayed. In the preceding image, student name and grade are displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship.

Create a view model

The instructors page shows data from three different tables. A view model is needed that includes three properties representing the three tables.

Create `Models/SchoolViewModels/InstructorIndexData.cs` with the following code:

```
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; }
    }
}
```

Scaffold Instructor pages

- Visual Studio
- Visual Studio Code

- Follow the instructions in Scaffold the student pages with the following exceptions:
  - Create a *Pages/Instructors* folder.
  - Use `Instructor` for the model class.
  - Use the existing context class instead of creating a new one.

Run the app and navigate to the Instructors page.

Update `Pages/Instructors/Index.cshtml.cs` with the following code:

```
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;  // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData InstructorData { get; set; }
        public int InstructorID { get; set; }
        public int CourseID { get; set; }

        public async Task OnGetAsync(int? id, int? courseID)
        {
            InstructorData = new InstructorIndexData();
            InstructorData.Instructors = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.Courses)
                    .ThenInclude(c => c.Department)
                .OrderBy(i => i.LastName)
                .ToListAsync();

            if (id != null)
            {
                InstructorID = id.Value;
                Instructor instructor = InstructorData.Instructors
                    .Where(i => i.ID == id.Value).Single();
                InstructorData.Courses = instructor.Courses;
            }

            if (courseID != null)
            {
                CourseID = courseID.Value;
                IEnumerable<Enrollment> Enrollments = await _context.Enrollments
                    .Where(x => x.CourseID == CourseID)
                    .Include(i=>i.Student)
                    .ToListAsync();
                InstructorData.Enrollments = Enrollments;
            }
        }
    }
}
```

The `OnGetAsync` method accepts optional route data for the ID of the selected instructor.

Examine the query in the `Pages/Instructors/Index.cshtml.cs` file:

```
InstructorData = new InstructorIndexData();
InstructorData.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.Courses)
        .ThenInclude(c => c.Department)
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

The code specifies eager loading for the following navigation properties:

- `Instructor.OfficeAssignment`
- `Instructor.Courses`
  - `Course.Department`

The following code executes when an instructor is selected, that is, `id != null`.

```
if (id != null)
{
    InstructorID = id.Value;
    Instructor instructor = InstructorData.Instructors
        .Where(i => i.ID == id.Value).Single();
    InstructorData.Courses = instructor.Courses;
}
```

The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is loaded with the `Course` entities from the selected instructor's `Courses` navigation property.

The `Where` method returns a collection. In this case, the filter select a single entity, so the `Single` method is called to convert the collection into a single `Instructor` entity. The `Instructor` entity provides access to the `Course` navigation property.

The Single method is used on a collection when the collection has only one item. The `Single` method throws an exception if the collection is empty or if there's more than one item. An alternative is SingleOrDefault, which returns a default value if the collection is empty. For this query, `null` in the default returned.

The following code populates the view model's `Enrollments` property when a course is selected:

```
if (courseID != null)
{
    CourseID = courseID.Value;
    IEnumerable<Enrollment> Enrollments = await _context.Enrollments
        .Where(x => x.CourseID == CourseID)
        .Include(i=>i.Student)
        .ToListAsync();
    InstructorData.Enrollments = Enrollments;
}
```

Update the instructors Index page

Update `Pages/Instructors/Index.cshtml` with the following code.

```
@page "{id:int?}"
@model ContosoUniversity.Pages.Instructors.IndexModel

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

<h2>Instructors</h2>
```

```
<p>
    <a asp-page="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.InstructorData.Instructors)
        {
            string selectedRow = "";
            if (item.ID == Model.InstructorID)
            {
                selectedRow = "table-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.Courses)
                        {
                            @course.CourseID @:  @course.Title <br />
                        }
                    }
                </td>
                <td>
                    <a asp-page="./Index" asp-route-id="@item.ID">Select</a> |
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@if (Model.InstructorData.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.InstructorData.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == Model.CourseID)
            {
                selectedRow = "table-success";
            }
            <tr class="@selectedRow">
                <td>
                    <a asp-page="./Index" asp-route-courseID="@item.CourseID">Select</a>
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}

@if (Model.InstructorData.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.InstructorData.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

The preceding code makes the following changes:

- Updates the `page` directive to `@page "{id:int?}"`. `"{id:int?}"` is a route template. The route template changes integer query strings in the URL to route data. For example, clicking on the **Select** link for an instructor with only the `@page` directive produces a URL like the following:

  `https://localhost:5001/Instructors?id=2`

  When the page directive is `@page "{id:int?}"`, the URL is: `https://localhost:5001/Instructors/2`

- Adds an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` isn't null. Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.

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

- Adds a **Courses** column that displays courses taught by each instructor. See Explicit line transition for more about this razor syntax.
- Adds code that dynamically adds `class="table-success"` to the `tr` element of the selected instructor and course. This sets a background color for the selected row using a Bootstrap class.

```
string selectedRow = "";
if (item.CourseID == Model.CourseID)
{
    selectedRow = "table-success";
}
<tr class="@selectedRow">
```

- Adds a new hyperlink labeled **Select**. This link sends the selected instructor's ID to the `Index` method and sets a background color.

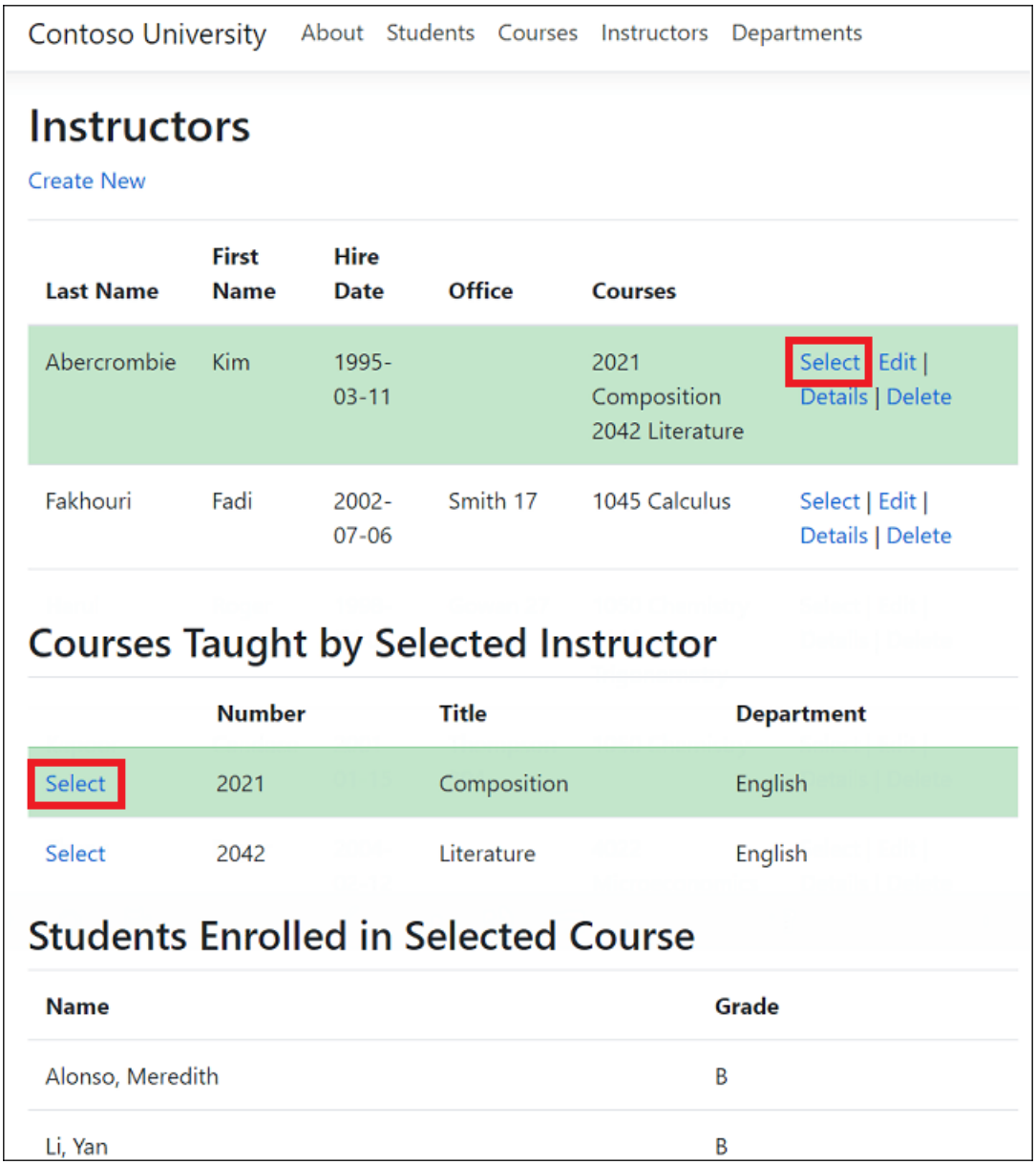
```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

- Adds a table of courses for the selected Instructor.
- Adds a table of student enrollments for the selected course.

Run the app and select the **Instructors** tab. The page displays the `Location` (office) from the related `OfficeAssignment` entity. If `OfficeAssignment` is null, an empty table cell is displayed.

Click on the **Select** link for an instructor. The row style changes and courses assigned to that instructor are displayed.

Select a course to see the list of enrolled students and their grades.



Next steps

The next tutorial shows how to update related data.

PREVIOUS TUTORIAL
NEXT TUTORIAL

This tutorial shows how to read and display related data. Related data is data that EF Core loads into navigation properties.

The following illustrations show the completed pages for this tutorial:

Contoso University About Students Courses Instructors Departments

Courses

Create New

Number	Title	Credits	Department	
1045	Calculus	4	Mathematics	Edit | Details | Delete
1050	Chemistry	3	Engineering	Edit | Details | Delete
2021	Composition	3	English	Edit | Details | Delete

Contoso University About Students Courses Instructors Departments

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

Students Enrolled in Selected Course

Name	Grade
Alonso, Meredith	B
Li, Yan	B

Eager, explicit, and lazy loading

There are several ways that EF Core can load related data into the navigation properties of an entity:

- Eager loading. Eager loading is when a query for one type of entity also loads related entities. When an entity is read, its related data is retrieved. This typically results in a single join query that retrieves all of the data that's needed. EF Core will issue multiple queries for some types of eager loading. Issuing

multiple queries can be more efficient than a giant single query. Eager loading is specified with 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

Eager loading sends multiple queries when a collection navigation is included:

- One query for the main query
- One query for each collection "edge" in the load tree.

- Separate queries with `Load`: The data can be retrieved in separate queries, and EF Core "fixes up" the navigation properties. "Fixes up" means that EF Core automatically populates the navigation properties. Separate queries with `Load` is more like explicit loading than eager loading.

```
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

Note: EF Core automatically fixes up navigation properties to any other entities that were previously loaded into the context instance. Even if the data for a navigation property is *not* explicitly included, the property may still be populated if some or all of the related entities were previously loaded.

- Explicit loading. When the entity is first read, related data isn't retrieved. Code must be written to retrieve the related data when it's needed. Explicit loading with separate queries results in multiple queries sent to the database. With explicit loading, the code specifies the navigation properties to be loaded. 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);
    }
}
```

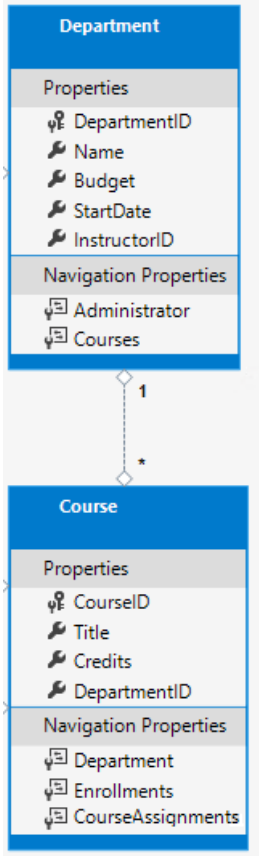
Query: all Department rows

Query: Course rows related to Department d

- Lazy loading. When the entity is first read, related data isn't retrieved. The first time a navigation property is accessed, the data required for that navigation property is automatically retrieved. A query is sent to the database each time a navigation property is accessed for the first time. Lazy loading can hurt performance, for example when developers use N+1 patterns, loading a parent and enumerating through children.

Create Course pages

The `Course` entity includes a navigation property that contains the related `Department` entity.



To display the name of the assigned department for a course:

- Load the related `Department` entity into the `Course.Department` navigation property.
- Get the name from the `Department` entity's `Name` property.

Scaffold Course pages

- Visual Studio
- Visual Studio Code

- Follow the instructions in Scaffold Student pages with the following exceptions:
  - Create a *Pages/Courses* folder.
  - Use `Course` for the model class.
  - Use the existing context class instead of creating a new one.
- Open `Pages/Courses/Index.cshtml.cs` and examine the `OnGetAsync` method. The scaffolding engine specified eager loading for the `Department` navigation property. The `Include` method specifies eager loading.
- Run the app and select the **Courses** link. The department column displays the `DepartmentID`, which isn't useful.

Display the department name

Update Pages/Courses/Index.cshtml.cs with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IList<Course> Courses { get; set; }

        public async Task OnGetAsync()
        {
            Courses = await _context.Courses
                .Include(c => c.Department)
                .AsNoTracking()
                .ToListAsync();
        }
    }
}
```

The preceding code changes the `Course` property to `Courses` and adds `AsNoTracking`. `AsNoTracking` improves performance because the entities returned are not tracked. The entities don't need to be tracked because they're not updated in the current context.

Update `Pages/Courses/Index.cshtml` with the following code.

```
@page
@model ContosoUniversity.Pages.Courses.IndexModel

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

<h1>Courses</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Courses)
{
        <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-page="./Edit" asp-route-id="@item.CourseID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.CourseID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.CourseID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

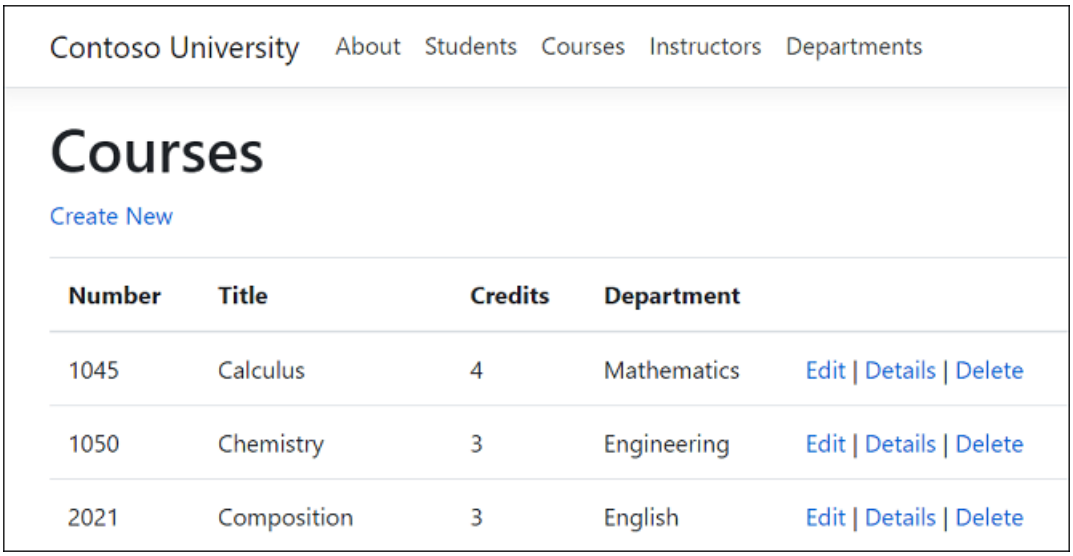
The following changes have been made to the scaffolded code:

- Changed the `Course` property name to `Courses`.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they're meaningless to end users. However, in this case the primary key is meaningful.
- 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:

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

Run the app and select the **Courses** tab to see the list with department names.



Loading related data with Select

The `OnGetAsync` method loads related data with the `Include` method. The `Select` method is an alternative that loads only the related data needed. For single items, like the `Department.Name` it uses a SQL INNER JOIN. For collections, it uses another database access, but so does the `Include` operator on collections.

The following code loads related data with the `Select` method:

```
public IList<CourseViewModel> CourseVM { get; set; }

public async Task OnGetAsync()
{
    CourseVM = await _context.Courses
            .Select(p => new CourseViewModel
            {
                CourseID = p.CourseID,
                Title = p.Title,
                Credits = p.Credits,
                DepartmentName = p.Department.Name
            }).ToListAsync();
}
```

The preceding code doesn't return any entity types, therefore no tracking is done. For more information about the EF tracking, see Tracking vs. No-Tracking Queries.

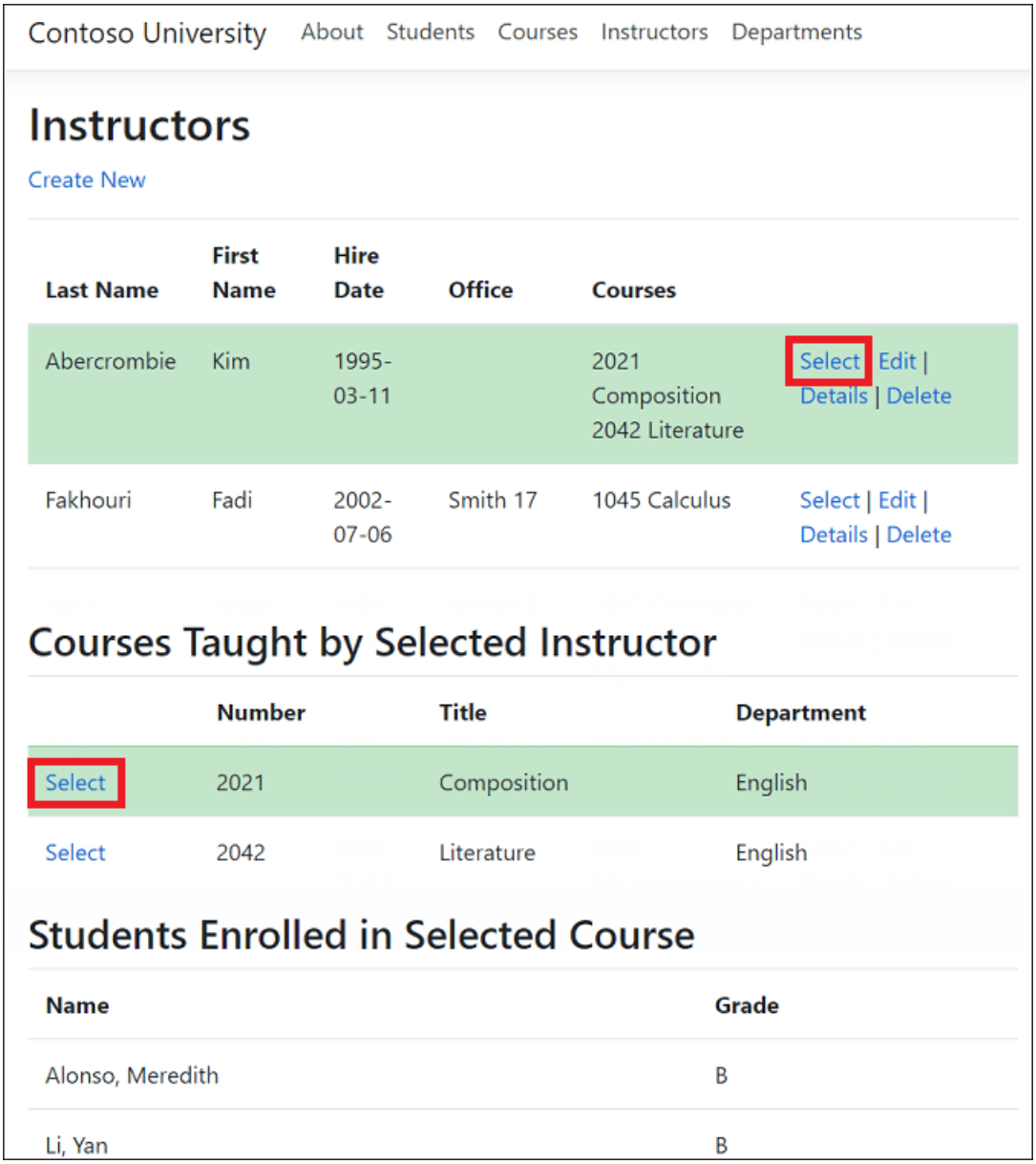
The `CourseViewModel`:

```
public class CourseViewModel
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public string DepartmentName { get; set; }
}
```

See IndexSelect.cshtml and IndexSelect.cshtml.cs for a complete example.

Create Instructor pages

This section scaffolds Instructor pages and adds related Courses and Enrollments to the Instructors Index page.



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

- The list of instructors displays related data from the `OfficeAssignment` entity (Office in the preceding image). The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. Eager loading is used for the `OfficeAssignment` entities. Eager loading is typically more efficient when the related data needs to be displayed. In this case, office assignments for the instructors are displayed.
- When the user selects an instructor, related `Course` entities are displayed. The `Instructor` and `Course` entities are in a many-to-many relationship. Eager loading is used for the `Course` entities and their related `Department` entities. In this case, separate queries might be more efficient because only courses for the selected instructor are needed. This example shows how to use eager loading for navigation properties in entities that are in navigation properties.
- When the user selects a course, related data from the `Enrollments` entity is displayed. In the preceding image, student name and grade are displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship.

Create a view model

The instructors page shows data from three different tables. A view model is needed that includes three properties representing the three tables.

Create `SchoolViewModels/InstructorIndexData.cs` with the following code:

```
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; }
    }
}
```

Scaffold Instructor pages

- Visual Studio
- Visual Studio Code

- Follow the instructions in Scaffold the student pages with the following exceptions:
  - Create a *Pages/Instructors* folder.
  - Use `Instructor` for the model class.
  - Use the existing context class instead of creating a new one.

To see what the scaffolded page looks like before you update it, run the app and navigate to the Instructors page.

Update `Pages/Instructors/Index.cshtml.cs` with the following code:

```
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;  // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData InstructorData { get; set; }
        public int InstructorID { get; set; }
        public int CourseID { get; set; }

        public async Task OnGetAsync(int? id, int? courseID)
        {
            InstructorData = new InstructorIndexData();
            InstructorData.Instructors = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                        .ThenInclude(i => i.Department)
                .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                        .ThenInclude(i => i.Enrollments)
                            .ThenInclude(i => i.Student)
                .AsNoTracking()
                .OrderBy(i => i.LastName)
                .ToListAsync();

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

            if (courseID != null)
            {
                CourseID = courseID.Value;
                var selectedCourse = InstructorData.Courses
                    .Where(x => x.CourseID == courseID).Single();
                InstructorData.Enrollments = selectedCourse.Enrollments;
            }
        }
    }
}
```

The `OnGetAsync` method accepts optional route data for the ID of the selected instructor.

Examine the query in the `Pages/Instructors/Index.cshtml.cs` file:

```
InstructorData.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

The code specifies eager loading for the following navigation properties:

- `Instructor.OfficeAssignment`
- `Instructor.CourseAssignments`
  - `CourseAssignments.Course`
    - `Course.Department`
    - `Course.Enrollments`
      - `Enrollment.Student`

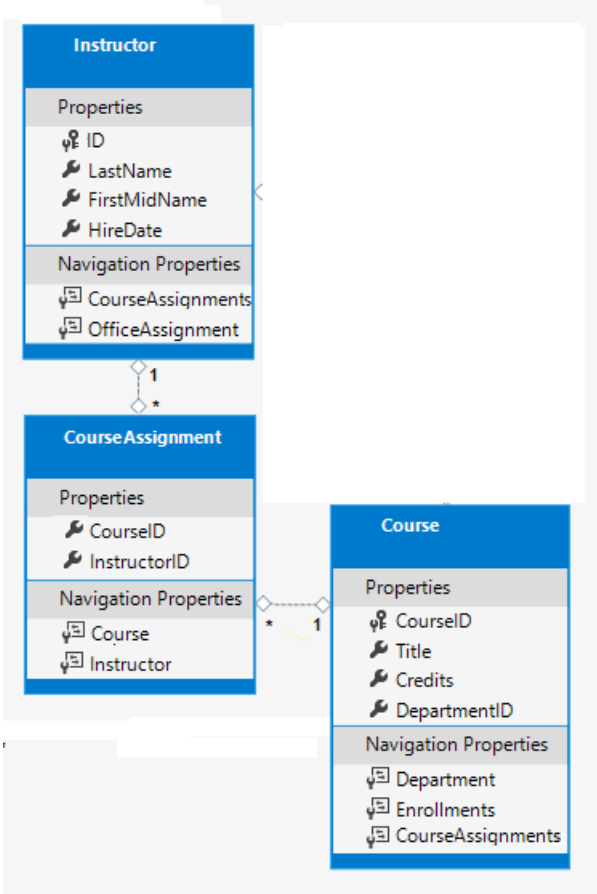
Notice the repetition of `Include` and `ThenInclude` methods for `CourseAssignments` and `Course`. This repetition is necessary to specify eager loading for two navigation properties of the `Course` entity.

The following code executes when an instructor is selected (`id != null`).

```
if (id != null)
{
    InstructorID = id.Value;
    Instructor instructor = InstructorData.Instructors
        .Where(i => i.ID == id.Value).Single();
    InstructorData.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is loaded with the `Course` entities from that instructor's `CourseAssignments` navigation property.

The `Where` method returns a collection. But in this case, the filter will select a single entity, so the `Single` method is called to convert the collection into a single `Instructor` entity. The `Instructor` entity provides access to the `CourseAssignments` property. `CourseAssignments` provides access to the related `Course` entities.



The `Single` method is used on a collection when the collection has only one item. The `Single` method throws an exception if the collection 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.

The following code populates the view model's `Enrollments` property when a course is selected:

```
if (courseID != null)
{
    CourseID = courseID.Value;
    var selectedCourse = InstructorData.Courses
        .Where(x => x.CourseID == courseID).Single();
    InstructorData.Enrollments = selectedCourse.Enrollments;
}
```

Update the instructors Index page

Update `Pages/Instructors/Index.cshtml` with the following code.

```
@page "{id:int?}"
@model ContosoUniversity.Pages.Instructors.IndexModel

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

<h2>Instructors</h2>

<p>
    <a asp-page="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.InstructorData.Instructors)
        {
            string selectedRow = "";
            if (item.ID == Model.InstructorID)
            {
                selectedRow = "table-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>
                <td>
                    <a asp-page="./Index" asp-route-id="@item.ID">Select</a> |
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@if (Model.InstructorData.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.InstructorData.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == Model.CourseID)
            {
                selectedRow = "table-success";
            }
            <tr class="@selectedRow">
                <td>
```

```
                    <a asp-page="./Index" asp-route-courseID="@item.CourseID">Select</a>
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}

@if (Model.InstructorData.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.InstructorData.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

The preceding code makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int?}"`. `"{id:int?}"` is a route template. The route template changes integer query strings in the URL to route data. For example, clicking on the **Select** link for an instructor with only the `@page` directive produces a URL like the following:

  `https://localhost:5001/Instructors?id=2`

  When the page directive is `@page "{id:int?}"`, the URL is:

  `https://localhost:5001/Instructors/2`

- Adds an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` isn't null. Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.

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

- Adds a **Courses** column that displays courses taught by each instructor. See Explicit line transition for

more about this razor syntax.

- Adds code that dynamically adds `class="table-success"` to the `tr` element of the selected instructor and course. This sets a background color for the selected row using a Bootstrap class.

```
string selectedRow = "";
if (item.CourseID == Model.CourseID)
{
    selectedRow = "table-success";
}
<tr class="@selectedRow">
```

- Adds a new hyperlink labeled **Select**. This link sends the selected instructor's ID to the `Index` method and sets a background color.

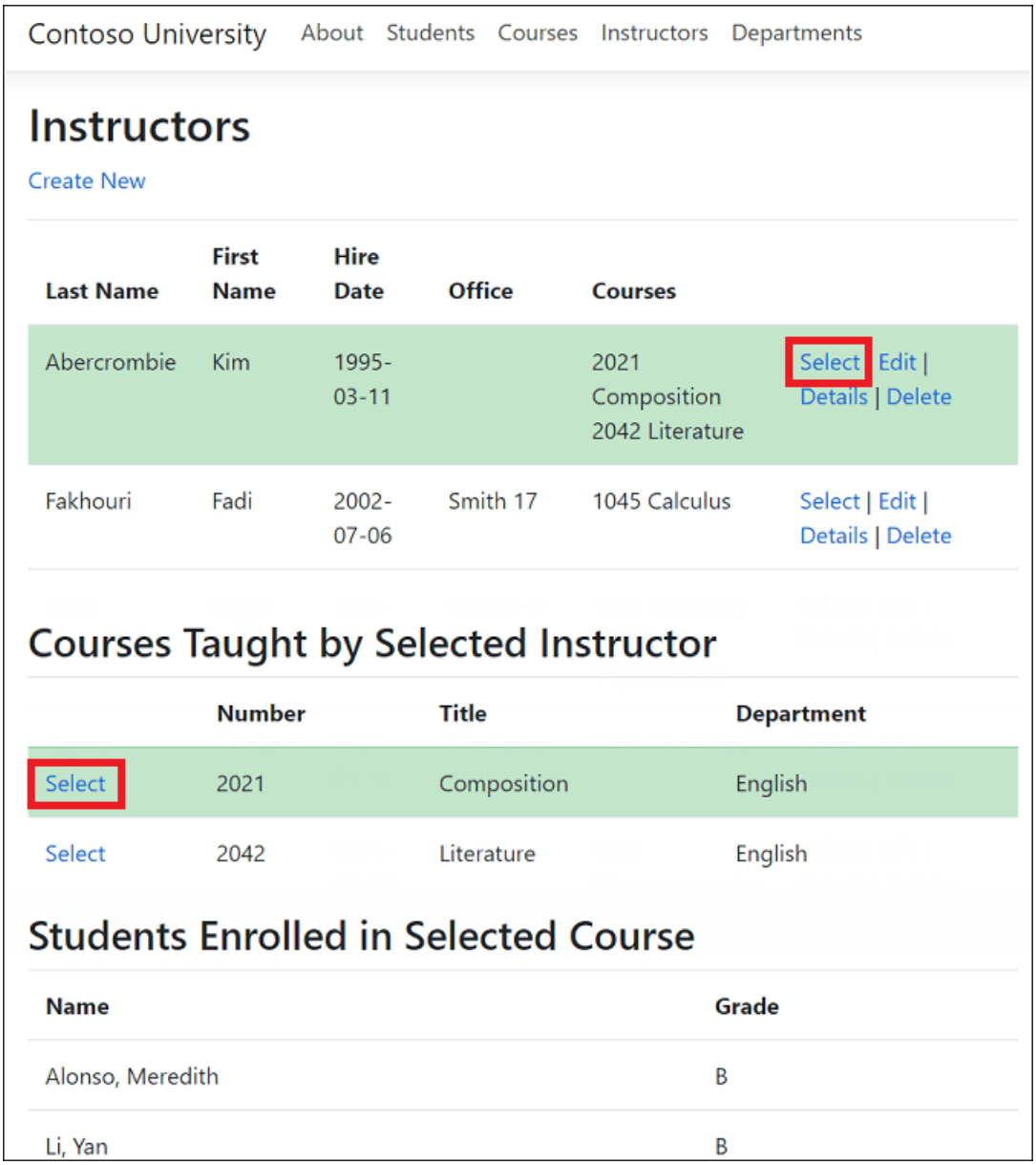
```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

- Adds a table of courses for the selected Instructor.
- Adds a table of student enrollments for the selected course.

Run the app and select the **Instructors** tab. The page displays the `Location` (office) from the related `OfficeAssignment` entity. If `OfficeAssignment` is null, an empty table cell is displayed.

Click on the **Select** link for an instructor. The row style changes and courses assigned to that instructor are displayed.

Select a course to see the list of enrolled students and their grades.



Using Single

The `Single` method can pass in the `Where` condition instead of calling the `Where` method separately:

```
public async Task OnGetAsync(int? id, int? courseID)
{
    InstructorData = new InstructorIndexData();

    InstructorData.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
            .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
                    .ThenInclude(i => i.Enrollments)
                        .ThenInclude(i => i.Student)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = InstructorData.Instructors.Single(
            i => i.ID == id.Value);
        InstructorData.Courses = instructor.CourseAssignments.Select(
            s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        InstructorData.Enrollments = InstructorData.Courses.Single(
            x => x.CourseID == courseID).Enrollments;
    }
}
```

Use of `Single` with a Where condition is a matter of personal preference. It provides no benefits over using the `Where` method.

Explicit loading

The current code specifies eager loading for `Enrollments` and `Students`:

```
InstructorData.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

Suppose users rarely want to see enrollments in a course. In that case, an optimization would be to only load the enrollment data if it's requested. In this section, the `OnGetAsync` is updated to use explicit loading of `Enrollments` and `Students`.

Update `Pages/Instructors/Index.cshtml.cs` with the following code.

```
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;  // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData InstructorData { get; set; }
        public int InstructorID { get; set; }
        public int CourseID { get; set; }

        public async Task OnGetAsync(int? id, int? courseID)
        {
            InstructorData = new InstructorIndexData();
            InstructorData.Instructors = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                        .ThenInclude(i => i.Department)
                //.Include(i => i.CourseAssignments)
                //    .ThenInclude(i => i.Course)
                //        .ThenInclude(i => i.Enrollments)
                //            .ThenInclude(i => i.Student)
                //.AsNoTracking()
                .OrderBy(i => i.LastName)
                .ToListAsync();

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

            if (courseID != null)
            {
                CourseID = courseID.Value;
                var selectedCourse = InstructorData.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();
                }
                InstructorData.Enrollments = selectedCourse.Enrollments;
            }
        }
    }
}
```

The preceding code drops the *ThenInclude* method calls for enrollment and student data. If a course is selected, the explicit loading code retrieves:

- The `Enrollment` entities for the selected course.

- The `Student` entities for each `Enrollment`.

Notice that the preceding code comments out `.AsNoTracking()`. Navigation properties can only be explicitly loaded for tracked entities.

Test the app. From a user's perspective, the app behaves identically to the previous version.

Next steps

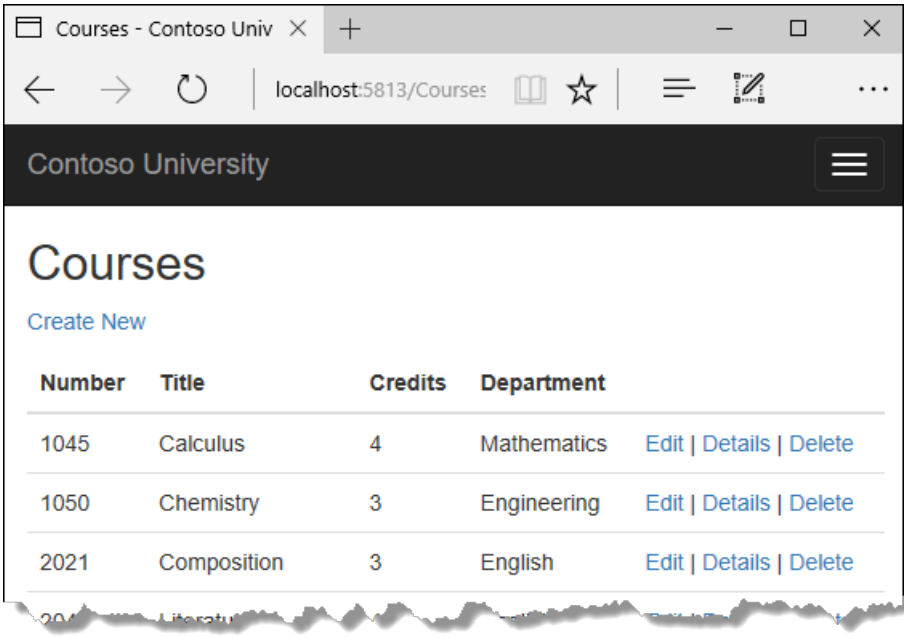
The next tutorial shows how to update related data.

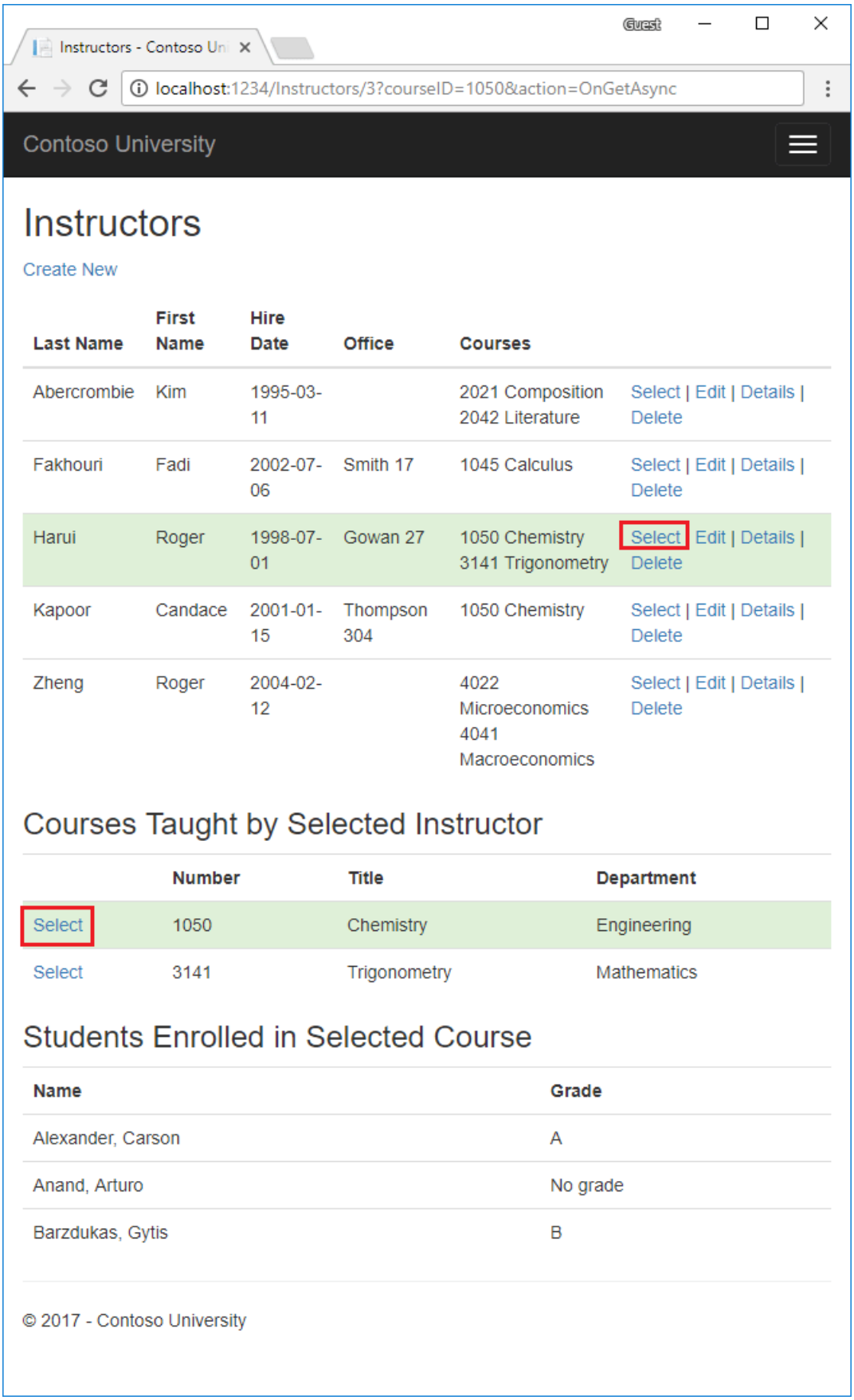
PREVIOUS TUTORIAL | NEXT TUTORIAL

In this tutorial, related data is read and displayed. Related data is data that EF Core loads into navigation properties.

If you run into problems you can't solve, download or view the completed app. Download instructions.

The following illustrations show the completed pages for this tutorial:





Eager, explicit, and lazy Loading of related data

There are several ways that EF Core can load related data into the navigation properties of an entity:

- Eager loading. Eager loading is when a query for one type of entity also loads related entities. When the entity is read, its related data is retrieved. This typically results in a single join query that retrieves all of the data that's needed. EF Core will issue multiple queries for some types of eager loading. Issuing multiple queries can be more efficient than was the case for some queries in EF6 where there was a

single query. Eager loading is specified with 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

Eager loading sends multiple queries when a collection navigation is included:

- One query for the main query
- One query for each collection "edge" in the load tree.

- Separate queries with `Load`: The data can be retrieved in separate queries, and EF Core "fixes up" the navigation properties. "fixes up" means that EF Core automatically populates the navigation properties. Separate queries with `Load` is more like explicit loading than eager loading.

```
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

Note: EF Core automatically fixes up navigation properties to any other entities that were previously loaded into the context instance. Even if the data for a navigation property is *not* explicitly included, the property may still be populated if some or all of the related entities were previously loaded.

- Explicit loading. When the entity is first read, related data isn't retrieved. Code must be written to retrieve the related data when it's needed. Explicit loading with separate queries results in multiple queries sent to the DB. With explicit loading, the code specifies the navigation properties to be loaded. 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. Lazy loading was added to EF Core in version 2.1. When the entity is first read, related data isn't retrieved. The first time a navigation property is accessed, the data required for that navigation property is automatically retrieved. A query is sent to the DB each time a navigation property is accessed for the first time.

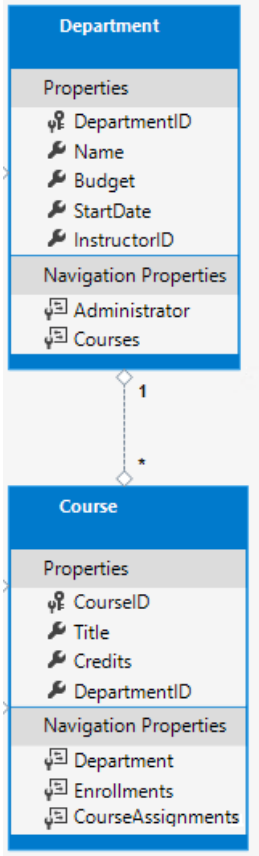
- The `Select` operator loads only the related data needed.

Create a Course page that displays department name

The Course entity includes a navigation property that contains the `Department` entity. The `Department` entity contains the department that the course is assigned to.

To display the name of the assigned department in a list of courses:

- Get the `Name` property from the `Department` entity.
- The `Department` entity comes from the `Course.Department` navigation property.



Scaffold the Course model

- Visual Studio
- Visual Studio Code

Follow the instructions in Scaffold the student model and use `Course` for the model class.

The preceding command scaffolds the `Course` model. Open the project in Visual Studio.

Open `Pages/Courses/Index.cshtml.cs` and examine the `OnGetAsync` method. The scaffolding engine specified eager loading for the `Department` navigation property. The `Include` method specifies eager loading.

Run the app and select the **Courses** link. The department column displays the `DepartmentID`, which isn't useful.

Update the `OnGetAsync` method with the following code:

```
public async Task OnGetAsync()
{
    Course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .ToListAsync();
}
```

The preceding code adds `AsNoTracking`. `AsNoTracking` improves performance because the entities returned are not tracked. The entities are not tracked because they're not updated in the current context.

Update `Pages/Courses/Index.cshtml` with the following highlighted markup:

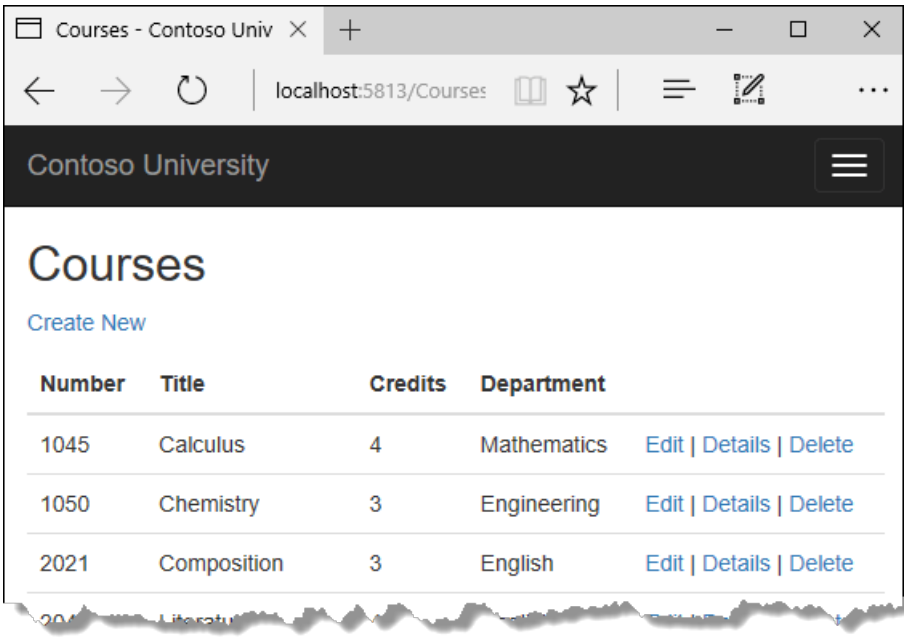
```
@page
@model ContosoUniversity.Pages.Courses.IndexModel
@{
    ViewData["Title"] = "Courses";
}

<h2>Courses</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Course)
        {
            <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-page="./Edit" asp-route-id="@item.CourseID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.CourseID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.CourseID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The following changes have been made 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're meaningless to end users. However, in this case the primary key is meaningful.
- 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:

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

Run the app and select the **Courses** tab to see the list with department names.



Loading related data with Select

The `OnGetAsync` method loads related data with the `Include` method:

```
public async Task OnGetAsync()
{
    Course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .ToListAsync();
}
```

The `Select` operator loads only the related data needed. For single items, like the `Department.Name` it uses a SQL INNER JOIN. For collections, it uses another database access, but so does the `Include` operator on collections.

The following code loads related data with the `Select` method:

```
public IList<CourseViewModel> CourseVM { get; set; }

public async Task OnGetAsync()
{
    CourseVM = await _context.Courses
            .Select(p => new CourseViewModel
            {
                CourseID = p.CourseID,
                Title = p.Title,
                Credits = p.Credits,
                DepartmentName = p.Department.Name
            }).ToListAsync();
}
```

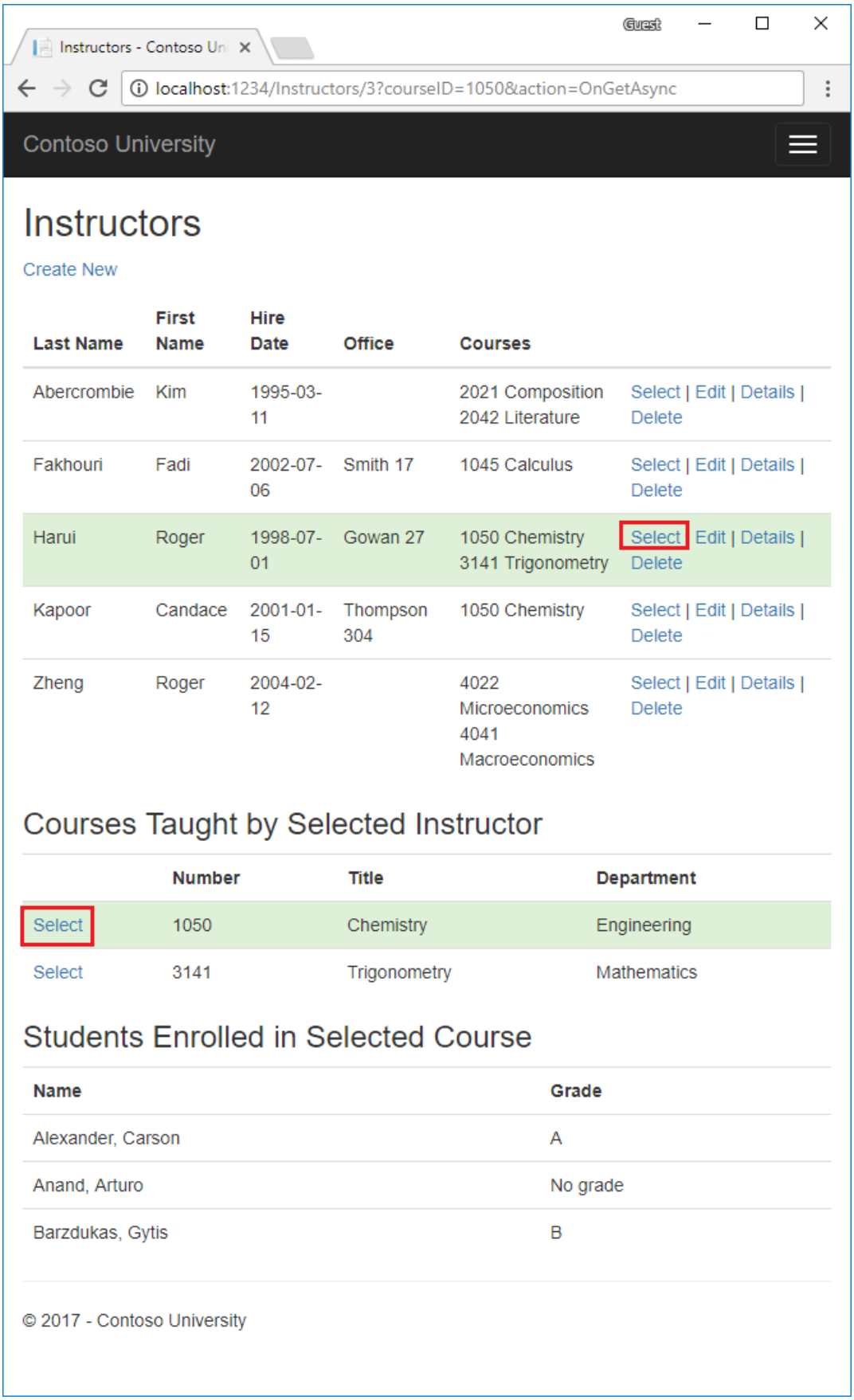
The `CourseViewModel`:

```
public class CourseViewModel
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public string DepartmentName { get; set; }
}
```

See IndexSelect.cshtml and IndexSelect.cshtml.cs for a complete example.

Create an Instructors page that shows Courses and Enrollments

In this section, the Instructors page is created.



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

- The list of instructors displays related data from the `OfficeAssignment` entity (Office in the preceding image). The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. Eager loading is used for the `OfficeAssignment` entities. Eager loading is typically more efficient when the related data needs to be displayed. In this case, office assignments for the instructors are displayed.
- When the user selects an instructor (Harui in the preceding image), related `Course` entities are displayed. The

`Instructor` and `Course` entities are in a many-to-many relationship. Eager loading is used for the `Course` entities and their related `Department` entities. In this case, separate queries might be more efficient because only courses for the selected instructor are needed. This example shows how to use eager loading for navigation properties in entities that are in navigation properties.

- When the user selects a course (Chemistry in the preceding image), related data from the `Enrollments` entity is displayed. In the preceding image, student name and grade are displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship.

Create a view model for the Instructor Index view

The instructors page shows data from three different tables. A view model is created that includes the three entities representing the three tables.

In the *SchoolViewModels* folder, create `InstructorIndexData.cs` with the following code:

```
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; }
    }
}
```

Scaffold the Instructor model

- Visual Studio
- Visual Studio Code

Follow the instructions in Scaffold the student model and use `Instructor` for the model class.

The preceding command scaffolds the `Instructor` model. Run the app and navigate to the instructors page.

Replace `Pages/Instructors/Index.cshtml.cs` with the following code:

```
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;  // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData Instructor { get; set; }
        public int InstructorID { get; set; }

        public async Task OnGetAsync(int? id)
        {
            Instructor = new InstructorIndexData();
            Instructor.Instructors = await _context.Instructors
                  .Include(i => i.OfficeAssignment)
                  .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                  .AsNoTracking()
                  .OrderBy(i => i.LastName)
                  .ToListAsync();

            if (id != null)
            {
                InstructorID = id.Value;
            }
        }
    }
}
```

The `OnGetAsync` method accepts optional route data for the ID of the selected instructor.

Examine the query in the `Pages/Instructors/Index.cshtml.cs` file:

```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

The query has two includes:

- `OfficeAssignment`: Displayed in the instructors view.
- `CourseAssignments`: Which brings in the courses taught.

Update the instructors Index page

Update `Pages/Instructors/Index.cshtml` with the following markup:

```
@page "{id:int?}"
@model ContosoUniversity.Pages.Instructors.IndexModel

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

<h2>Instructors</h2>

<p>
    <a asp-page="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.Instructor.Instructors)
        {
            string selectedRow = "";
            if (item.ID == Model.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>
                <td>
                    <a asp-page="./Index" asp-route-id="@item.ID">Select</a> |
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding markup makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int?}"`. `"{id:int?}"` is a route template. The route template changes integer query strings in the URL to route data. For example, clicking on the **Select** link for an instructor with only the `@page` directive produces a URL like the following:

  `http://localhost:1234/Instructors?id=2`

  When the page directive is `@page "{id:int?}"`, the previous URL is:

  `http://localhost:1234/Instructors/2`

- Page title is **Instructors**.

- Added an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` isn't null. Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.

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

- Added a **Courses** column that displays courses taught by each instructor. See Explicit line transition for more about this razor syntax.

- 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.

  ```
  string selectedRow = "";
  if (item.CourseID == Model.CourseID)
  {
      selectedRow = "success";
  }
  <tr class="@selectedRow">
  ```

- Added a new hyperlink labeled **Select**. This link sends the selected instructor's ID to the `Index` method and sets a background color.

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

Run the app and select the **Instructors** tab. The page displays the `Location` (office) from the related `OfficeAssignment` entity. If OfficeAssignment` is null, an empty table cell is displayed.

Click on the **Select** link. The row style changes.

Add courses taught by selected instructor

Update the `OnGetAsync` method in `Pages/Instructors/Index.cshtml.cs` with the following code:

```
public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();
    Instructor.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

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

    if (courseID != null)
    {
        CourseID = courseID.Value;
        Instructor.Enrollments = Instructor.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }
}
```

Add `public int CourseID { get; set; }`

```
public class IndexModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public IndexModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    public InstructorIndexData Instructor { get; set; }
    public int InstructorID { get; set; }
    public int CourseID { get; set; }

    public async Task OnGetAsync(int? id, int? courseID)
    {
        Instructor = new InstructorIndexData();
        Instructor.Instructors = await _context.Instructors
              .Include(i => i.OfficeAssignment)
              .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
                    .ThenInclude(i => i.Department)
              .AsNoTracking()
              .OrderBy(i => i.LastName)
              .ToListAsync();

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

        if (courseID != null)
        {
            CourseID = courseID.Value;
            Instructor.Enrollments = Instructor.Courses.Where(
                x => x.CourseID == courseID).Single().Enrollments;
        }
    }
```

Examine the updated query:

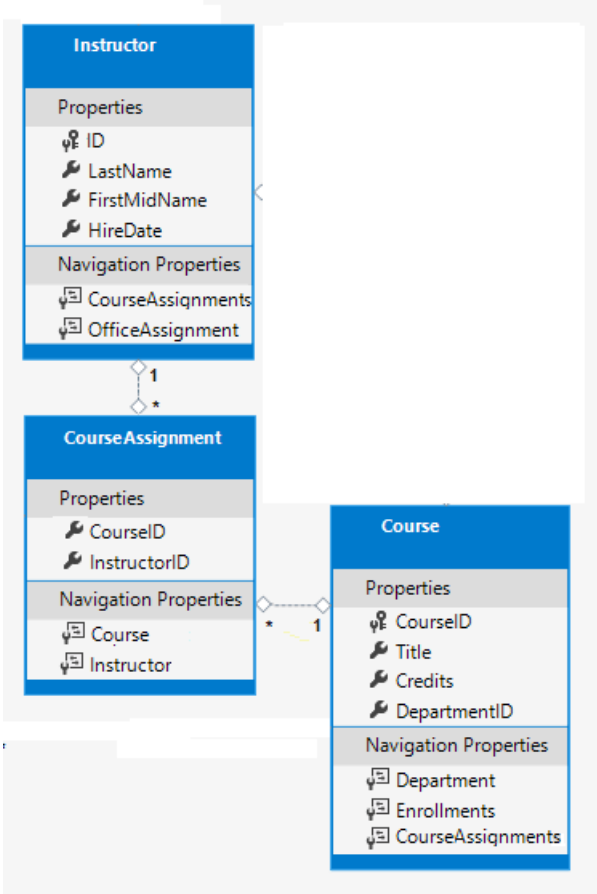
```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

The preceding query adds the `Department` entities.

The following code executes when an instructor is selected (`id != null`). The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is loaded with the `Course` entities from that instructor's `CourseAssignments` navigation property.

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

The `Where` method returns a collection. In the preceding `Where` method, only a single `Instructor` entity is returned. The `Single` method converts the collection into a single `Instructor` entity. The `Instructor` entity provides access to the `CourseAssignments` property. `CourseAssignments` provides access to the related `Course` entities.



The `Single` method is used on a collection when the collection has only one item. The `Single` method throws an exception if the collection 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. Using `SingleOrDefault` on an empty collection:

- Results in an exception (from trying to find a `Courses` property on a null reference).
- The exception message would less clearly indicate the cause of the problem.

The following code populates the view model's `Enrollments` property when a course is selected:

```
if (courseID != null)
{
    CourseID = courseID.Value;
    Instructor.Enrollments = Instructor.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

Add the following markup to the end of the `Pages/Instructors/Index.cshtml` Razor Page:

```
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@if (Model.Instructor.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.Instructor.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == Model.CourseID)
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    <a asp-page="./Index" asp-route-courseID="@item.CourseID">Select</a>
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}
```

The preceding markup displays a list of courses related to an instructor when an instructor is selected.

Test the app. Click on a **Select** link on the instructors page.

Show student data

In this section, the app is updated to show the student data for a selected course.

Update the query in the `OnGetAsync` method in `Pages/Instructors/Index.cshtml.cs` with the following code:

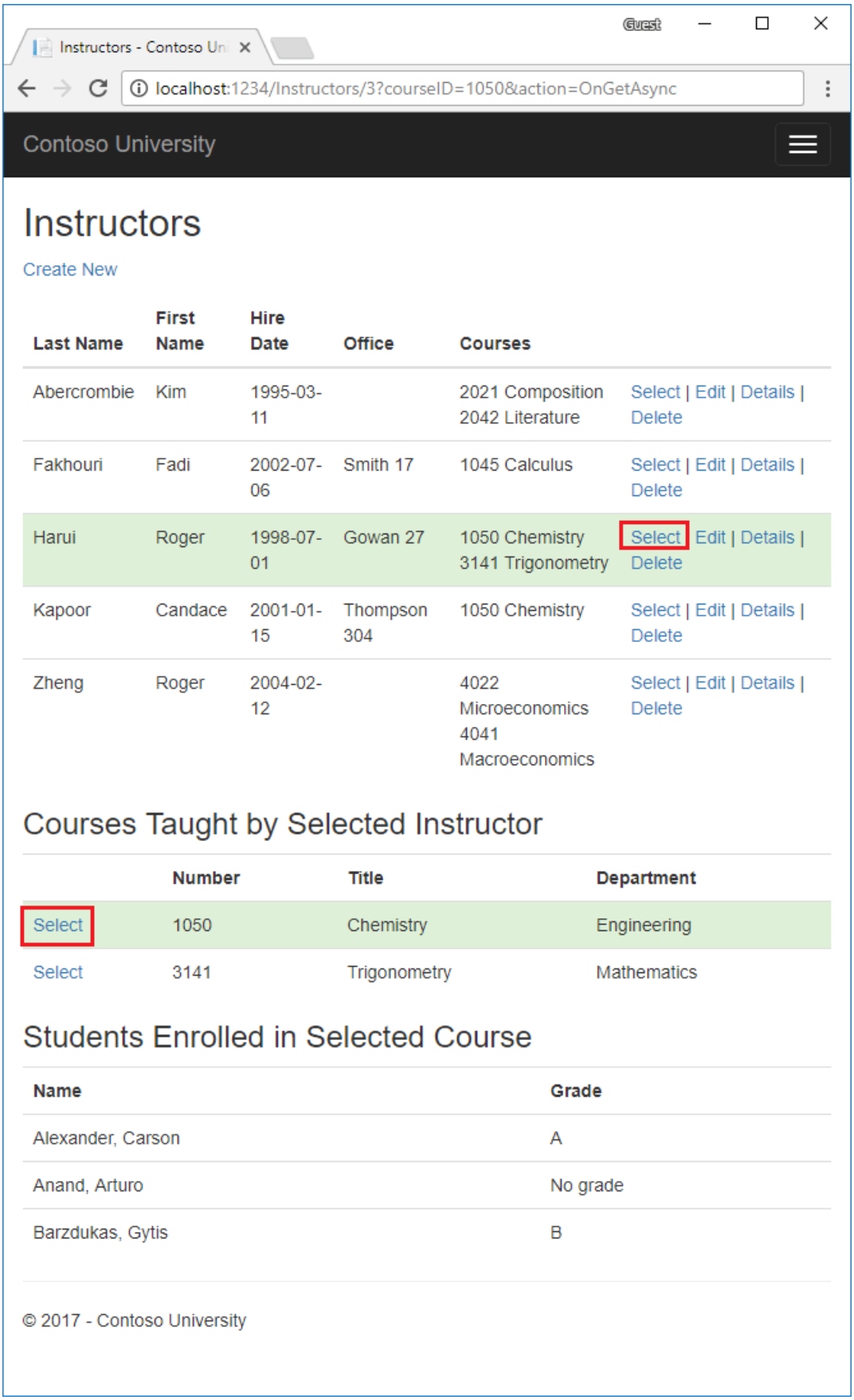
```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

Update `Pages/Instructors/Index.cshtml`. Add the following markup to the end of the file:

```
@if (Model.Instructor.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.Instructor.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

The preceding markup displays a list of the students who are enrolled in the selected course.

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



Using Single

The `Single` method can pass in the `Where` condition instead of calling the `Where` method separately:

```
public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();

    Instructor.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
            .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
                    .ThenInclude(i => i.Enrollments)
                        .ThenInclude(i => i.Student)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

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

    if (courseID != null)
    {
        CourseID = courseID.Value;
        Instructor.Enrollments = Instructor.Courses.Single(
            x => x.CourseID == courseID).Enrollments;
    }
}
```

The preceding `Single` approach provides no benefits over using `Where`. Some developers prefer the `Single` approach style.

Explicit loading

The current code specifies eager loading for `Enrollments` and `Students`:

```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

Suppose users rarely want to see enrollments in a course. In that case, an optimization would be to only load the enrollment data if it's requested. In this section, the `OnGetAsync` is updated to use explicit loading of `Enrollments` and `Students`.

Update the `OnGetAsync` with the following code:

```
public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();
    Instructor.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
            //.Include(i => i.CourseAssignments)
            //    .ThenInclude(i => i.Course)
            //        .ThenInclude(i => i.Enrollments)
            //            .ThenInclude(i => i.Student)
         // .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();


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

    if (courseID != null)
    {
        CourseID = courseID.Value;
        var selectedCourse = Instructor.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();
        }
        Instructor.Enrollments = selectedCourse.Enrollments;
    }
}
```

The preceding code drops the *ThenInclude* method calls for enrollment and student data. If a course is selected, the highlighted code retrieves:

- The `Enrollment` entities for the selected course.
- The `Student` entities for each `Enrollment`.

Notice the preceding code comments out `.AsNoTracking()`. Navigation properties can only be explicitly loaded for tracked entities.

Test the app. From a users perspective, the app behaves identically to the previous version.

The next tutorial shows how to update related data.

Additional resources

- YouTube version of this tutorial (part1)
- YouTube version of this tutorial (part2)

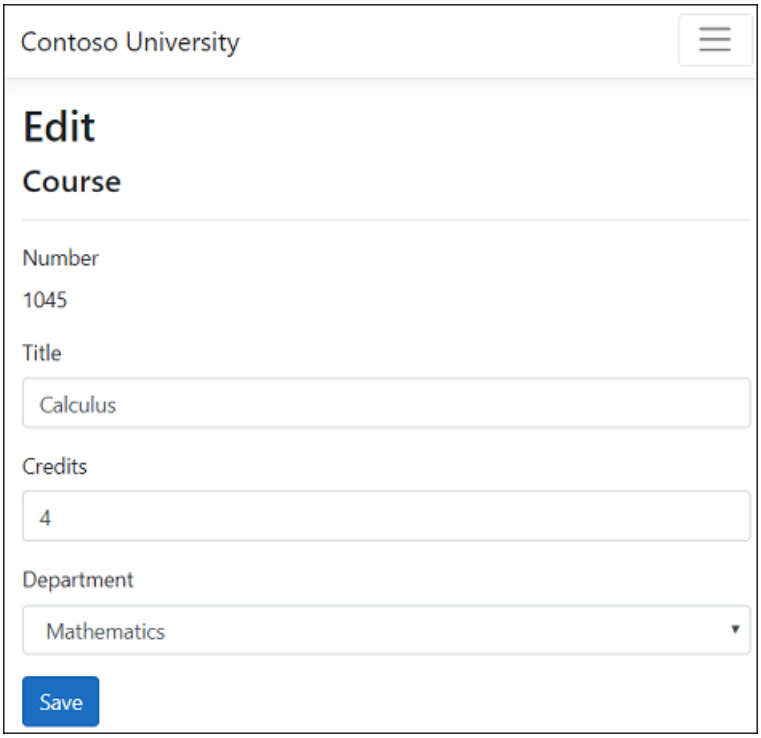
Part 7, Razor Pages with EF Core in ASP.NET Core - Update Related Data

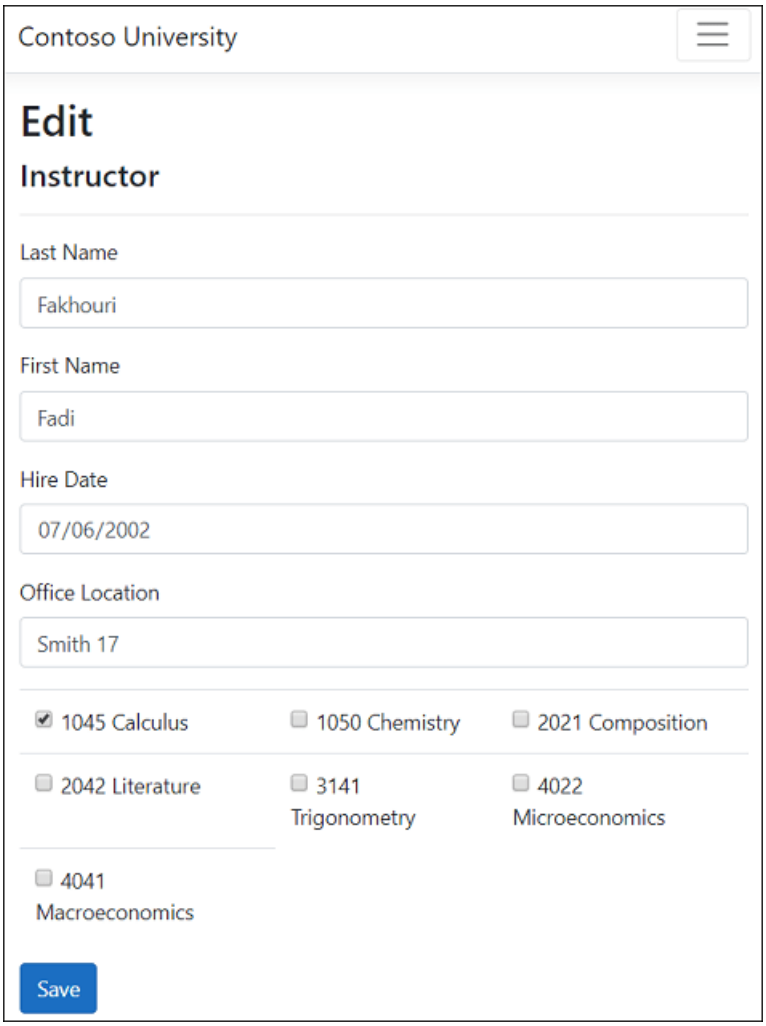
By Tom Dykstra, Jon P Smith, and Rick Anderson

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial shows how to update related data. The following illustrations show some of the completed pages.





Update the Course Create and Edit pages

The scaffolded code for the Course Create and Edit pages has a Department drop-down list that shows `DepartmentID`, an `int`. The drop-down should show the Department name, so both of these pages need a list of department names. To provide that list, use a base class for the Create and Edit pages.

Create a base class for Course Create and Edit

Create a `Pages/Courses/DepartmentNamePageModel.cs` file with the following code:

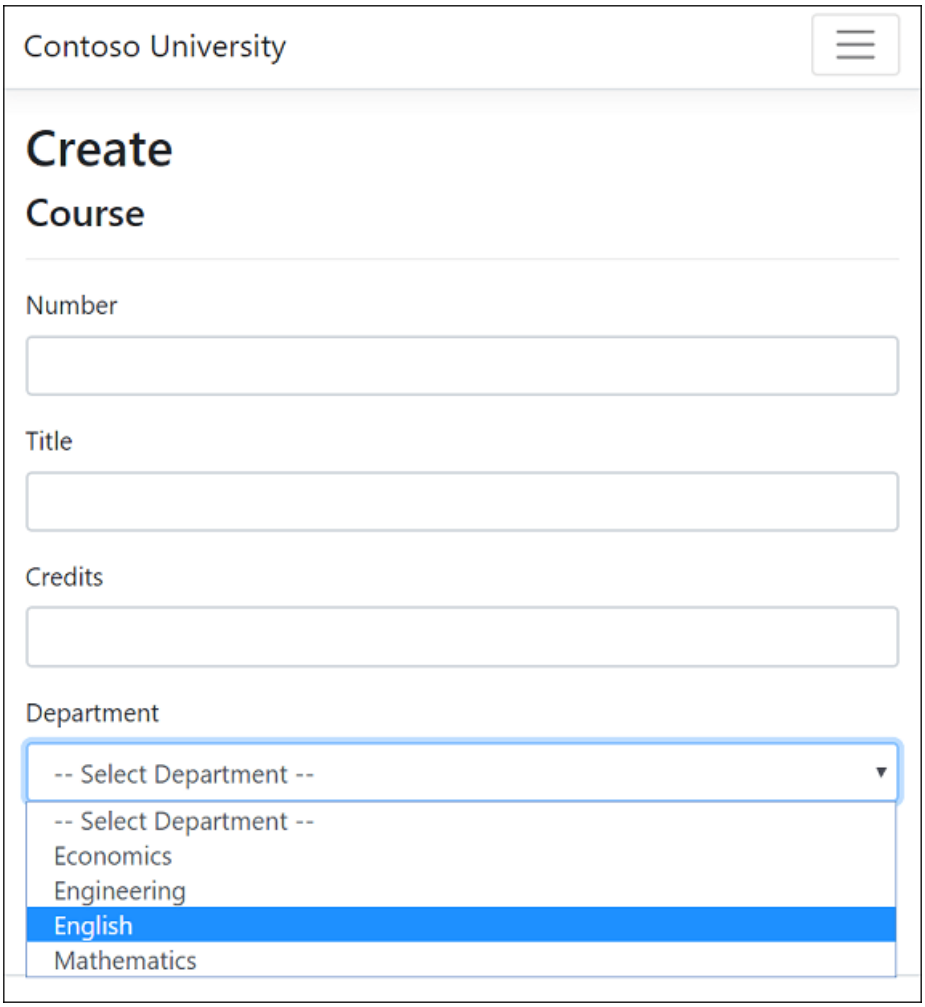
```
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace ContosoUniversity.Pages.Courses
{
    public class DepartmentNamePageModel : PageModel
    {
        public SelectList DepartmentNameSL { get; set; }

        public void PopulateDepartmentsDropDownList(SchoolContext _context,
            object selectedDepartment = null)
        {
            var departmentsQuery = from d in _context.Departments
                                   orderby d.Name // Sort by name.
                                   select d;

            DepartmentNameSL = new SelectList(departmentsQuery.AsNoTracking(),
                        "DepartmentID", "Name", selectedDepartment);
        }
    }
}
```

The preceding code creates a SelectList to contain the list of department names. If `selectedDepartment` is specified, that department is selected in the `SelectList`.

The Create and Edit page model classes will derive from `DepartmentNamePageModel`.

Update the Course Create page model

A Course is assigned to a Department. The base class for the Create and Edit pages provides a `SelectList` for selecting the department. The drop-down list that uses the `SelectList` sets the `Course.DepartmentID` foreign key (FK) property. EF Core uses the `Course.DepartmentID` FK to load the `Department` navigation property.



Update `Pages/Courses/Create.cshtml.cs` with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class CreateModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            PopulateDepartmentsDropDownList(_context);
            return Page();
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            var emptyCourse = new Course();

            if (await TryUpdateModelAsync<Course>(
                 emptyCourse,
                 "course",   // Prefix for form value.
                 s => s.CourseID, s => s.DepartmentID, s => s.Title, s => s.Credits))
            {
                _context.Courses.Add(emptyCourse);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, emptyCourse.DepartmentID);
            return Page();
        }
      }
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

The preceding code:

- Derives from `DepartmentNamePageModel`.
- Uses TryUpdateModelAsync to prevent overposting.
- Removes `ViewData["DepartmentID"]`. The `DepartmentNameSL` `SelectList` is a strongly typed model and will be used by the Razor page. Strongly typed models are preferred over weakly typed. For more information, see Weakly typed data (ViewData and ViewBag).

Update the Course Create Razor page

Update `Pages/Courses/Create.cshtml` with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.CreateModel
@{
    ViewData["Title"] = "Create Course";
}
<h2>Create</h2>
<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <input asp-for="Course.CourseID" class="form-control" />
                <span asp-validation-for="Course.CourseID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL">
                    <option value="">-- Select Department --</option>
                </select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code makes the following changes:

- Changes the caption from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).
- Adds the "Select Department" option. This change renders "Select Department" in the drop-down when no department has been selected yet, rather than the first department.
- Adds a validation message when the department isn't selected.

The Razor Page uses the Select Tag Helper:

```
<div class="form-group">
    <label asp-for="Course.Department" class="control-label"></label>
    <select asp-for="Course.DepartmentID" class="form-control"
            asp-items="@Model.DepartmentNameSL">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="Course.DepartmentID" class="text-danger" />
</div>
```

Test the Create page. The Create page displays the department name rather than the department ID.

Update the Course Edit page model

Update `Pages/Courses/Edit.cshtml.cs` with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class EditModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

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

            Course = await _context.Courses
                .Include(c => c.Department).FirstOrDefaultAsync(m => m.CourseID == id);

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

            // Select current DepartmentID.
            PopulateDepartmentsDropDownList(_context, Course.DepartmentID);
            return Page();
        }

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

            var courseToUpdate = await _context.Courses.FindAsync(id);

            if (courseToUpdate == null)
            {
                return NotFound();
```

```
            }

            if (await TryUpdateModelAsync<Course>(
                 courseToUpdate,
                 "course",   // Prefix for form value.
                   c => c.Credits, c => c.DepartmentID, c => c.Title))
            {
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, courseToUpdate.DepartmentID);
            return Page();
        }
    }
}
```

The changes are similar to those made in the Create page model. In the preceding code, `PopulateDepartmentsDropDownList` passes in the department ID, which selects that department in the drop-down list.

Update the Course Edit Razor page

Update `Pages/Courses/Edit.cshtml` with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.EditModel

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

<h2>Edit</h2>

<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Course.CourseID" />
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <div>@Html.DisplayFor(model => model.Course.CourseID)</div>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL"></select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code makes the following changes:

- Displays the course ID. Generally the Primary Key (PK) of an entity isn't displayed. PKs are usually meaningless to users. In this case, the PK is the course number.
- Changes the caption for the Department drop-down from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL`, which is in the base class.

The page contains a hidden field (`<input type="hidden">`) for the course number. Adding a `<label>` tag helper with `asp-for="Course.CourseID"` doesn't eliminate the need for the hidden field. `<input type="hidden">` is required for the course number to be included in the posted data when the user selects **Save**.

Update the Course page models

AsNoTracking can improve performance when tracking isn't required.

Update `Pages/Courses/Delete.cshtml.cs` and `Pages/Courses/Details.cshtml.cs` by adding `AsNoTracking` to the `OnGetAsync` methods:

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

    Course = await _context.Courses
        .AsNoTracking()
        .Include(c => c.Department)
        .FirstOrDefaultAsync(m => m.CourseID == id);

    if (Course == null)
    {
        return NotFound();
    }
    return Page();
}
```

Update the Course Razor pages

Update `Pages/Courses/Delete.cshtml` with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.DeleteModel

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

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Department.Name)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Course.CourseID" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Make the same changes to the Details page.

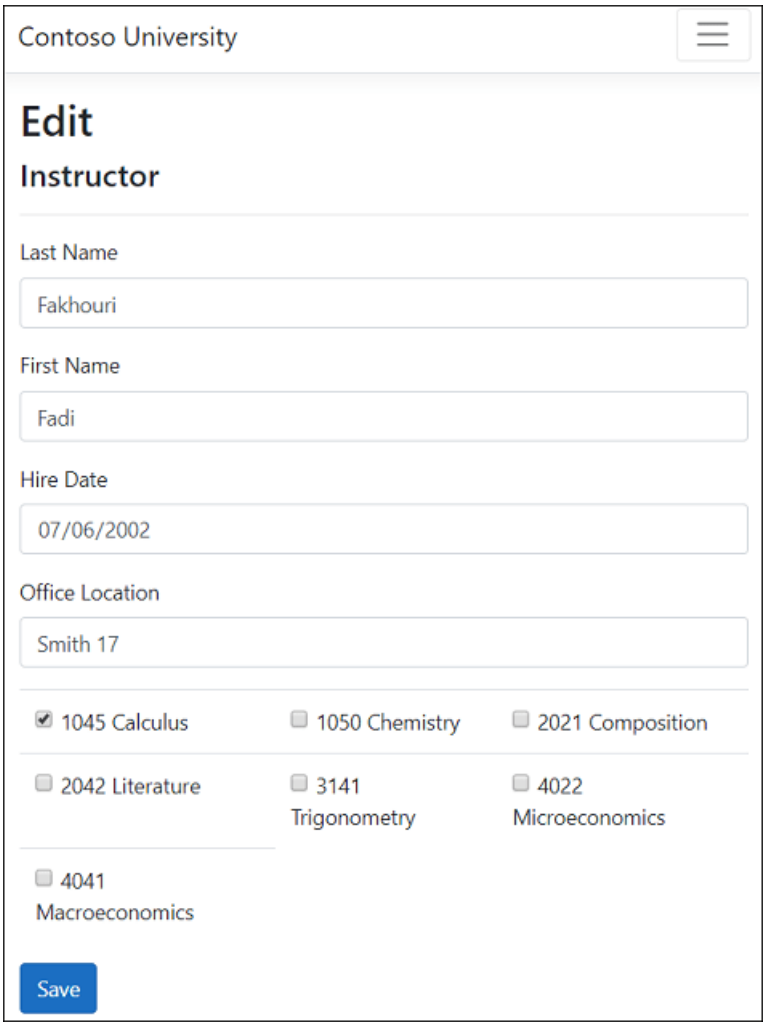
```
@page
@model ContosoUniversity.Pages.Courses.DetailsModel

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

<h2>Details</h2>

<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Department.Name)
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Course.CourseID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

Test the Course pages

Test the create, edit, details, and delete pages.

Update the instructor Create and Edit pages

Instructors may teach any number of courses. The following image shows the instructor Edit page with an array of course checkboxes.



The checkboxes enable changes to courses an instructor is assigned to. A checkbox is displayed for every course in the database. Courses that the instructor is assigned to are selected. The user can select or clear checkboxes to change course assignments. If the number of courses were much greater, a different UI might work better. But the method of managing a many-to-many relationship shown here wouldn't change. To create or delete relationships, you manipulate a join entity.

Create a class for assigned courses data

Create `Models/SchoolViewModels/AssignedCourseData.cs` with the following code:

```
namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

The `AssignedCourseData` class contains data to create the checkboxes for courses assigned to an instructor.

Create an Instructor page model base class

Create the `Pages/Instructors/InstructorCoursesPageModel.cs` base class:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using System.Linq;

namespace ContosoUniversity.Pages.Instructors
{
    public class InstructorCoursesPageModel : PageModel
    {
        public List<AssignedCourseData> AssignedCourseDataList;

        public void PopulateAssignedCourseData(SchoolContext context,
                                               Instructor instructor)
        {
            var allCourses = context.Courses;
            var instructorCourses = new HashSet<int>(
                instructor.Courses.Select(c => c.CourseID));
            AssignedCourseDataList = new List<AssignedCourseData>();
            foreach (var course in allCourses)
            {
                AssignedCourseDataList.Add(new AssignedCourseData
                {
                    CourseID = course.CourseID,
                    Title = course.Title,
                    Assigned = instructorCourses.Contains(course.CourseID)
                });
            }
        }
    }
}
```

The `InstructorCoursesPageModel` is the base class for the Edit and Create page models. `PopulateAssignedCourseData` reads all `Course` entities to populate `AssignedCourseDataList`. For each course, the code sets the `CourseID`, title, and whether or not the instructor is assigned to the course. A HashSet is used for efficient lookups.

Handle office location

Another relationship the edit page has to handle is the one-to-zero-or-one relationship that the Instructor entity has with the `OfficeAssignment` entity. The instructor edit code must handle the following scenarios:

- If the user clears the office assignment, delete the `OfficeAssignment` entity.
- If the user enters an office assignment and it was empty, create a new `OfficeAssignment` entity.
- If the user changes the office assignment, update the `OfficeAssignment` entity.

Update the Instructor Edit page model

Update `Pages/Instructors/Edit.cshtml.cs` with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class EditModel : InstructorCoursesPageModel
    {
```

```
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Instructor Instructor { get; set; }

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

        Instructor = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .Include(i => i.Courses)
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.ID == id);

        if (Instructor == null)
        {
            return NotFound();
        }
        PopulateAssignedCourseData(_context, Instructor);
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id, string[] selectedCourses)
    {
        if (id == null)
        {
            return NotFound();
        }

        var instructorToUpdate = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .Include(i => i.Courses)
            .FirstOrDefaultAsync(s => s.ID == id);

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

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            if (String.IsNullOrWhiteSpace(
                instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            UpdateInstructorCourses(selectedCourses, instructorToUpdate);
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
        UpdateInstructorCourses(selectedCourses, instructorToUpdate);
        PopulateAssignedCourseData(_context, instructorToUpdate);
        return Page();
    }

    public void UpdateInstructorCourses(string[] selectedCourses
```

```
        public void UpdateInstructorCourses(string[] selectedCourses,
                                            Instructor instructorToUpdate)
        {
            if (selectedCourses == null)
            {
                instructorToUpdate.Courses = new List<Course>();
                return;
            }

            var selectedCoursesHS = new HashSet<string>(selectedCourses);
            var instructorCourses = new HashSet<int>
                (instructorToUpdate.Courses.Select(c => c.CourseID));
            foreach (var course in _context.Courses)
            {
                if (selectedCoursesHS.Contains(course.CourseID.ToString()))
                {
                    if (!instructorCourses.Contains(course.CourseID))
                    {
                        instructorToUpdate.Courses.Add(course);
                    }
                }
                else
                {
                    if (instructorCourses.Contains(course.CourseID))
                    {
                        var courseToRemove = instructorToUpdate.Courses.Single(
                                                c => c.CourseID == course.CourseID);
                        instructorToUpdate.Courses.Remove(courseToRemove);
                    }
                }
            }
        }
    }
}
```

The preceding code:

- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment` and `Courses` navigation properties.
- Updates the retrieved `Instructor` entity with values from the model binder. TryUpdateModelAsync prevents overposting.
- If the office location is blank, sets `Instructor.OfficeAssignment` to null. When `Instructor.OfficeAssignment` is null, the related row in the `OfficeAssignment` table is deleted.
- Calls `PopulateAssignedCourseData` in `OnGetAsync` to provide information for the checkboxes using the `AssignedCourseData` view model class.
- Calls `UpdateInstructorCourses` in `OnPostAsync` to apply information from the checkboxes to the Instructor entity being edited.
- Calls `PopulateAssignedCourseData` and `UpdateInstructorCourses` in `OnPostAsync` if TryUpdateModelAsync fails. These method calls restore the assigned course data entered on the page when it is redisplayed with an error message.

Since the Razor page doesn't have a collection of Course entities, the model binder can't automatically update the `Courses` navigation property. Instead of using the model binder to update the `Courses` navigation property, that's done in the new `UpdateInstructorCourses` method. Therefore you need to exclude the `Courses` property from model binding. This doesn't require any change to the code that calls TryUpdateModelAsync because you're using the overload with declared properties and `Courses` isn't in the include list.

If no checkboxes were selected, the code in `UpdateInstructorCourses` initializes the `instructorToUpdate.Courses` with an empty collection and returns:

```
if (selectedCourses == null)
{
    instructorToUpdate.Courses = new List<Course>();
    return;
}
```

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 page. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the checkbox for a course is selected but the course is ***not*** in the `Instructor.Courses` navigation property, the course is added to the collection in the navigation property.

```
if (selectedCoursesHS.Contains(course.CourseID.ToString()))
{
    if (!instructorCourses.Contains(course.CourseID))
    {
        instructorToUpdate.Courses.Add(course);
    }
}
```

If the checkbox for a course is ***not*** selected, but the course is in the `Instructor.Courses` navigation property, the course is removed from the navigation property.

```
else
{
    if (instructorCourses.Contains(course.CourseID))
    {
        var courseToRemove = instructorToUpdate.Courses.Single(
                                        c => c.CourseID == course.CourseID);
        instructorToUpdate.Courses.Remove(courseToRemove);
    }
}
```

Update the Instructor Edit Razor page

Update `Pages/Instructors/Edit.cshtml` with the following code:

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
```

```
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="table">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    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>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code creates an HTML table that has three columns. Each column has a checkbox and a caption containing the course number and title. The checkboxes all have the same name ("selectedCourses"). Using the same name informs the model binder to treat them as a group. The value attribute of each checkbox is set to `CourseID`. When the page is posted, the model binder passes an array that consists of the `CourseID` values for only the checkboxes that are selected.

When the checkboxes are initially rendered, courses assigned to the instructor are selected.

Note: The approach taken here to edit instructor course data works well when there's a limited number of courses. For collections that are much larger, a different UI and a different updating method would be more useable and efficient.

Run the app and test the updated Instructors Edit page. Change some course assignments. The changes are reflected on the Index page.

Update the Instructor Create page

Update the Instructor Create page model and with code similar to the Edit page:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class CreateModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;
        private readonly ILogger<InstructorCoursesPageModel> _logger;

        public CreateModel(SchoolContext context,
                           ILogger<InstructorCoursesPageModel> logger)
        {
            _context = context;
            _logger = logger;
        }

        public IActionResult OnGet()
        {
            var instructor = new Instructor();
            instructor.Courses = new List<Course>();

            // Provides an empty collection for the foreach loop
            // foreach (var course in Model.AssignedCourseDataList)
            // in the Create Razor page.
            PopulateAssignedCourseData(_context, instructor);
            return Page();
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnPostAsync(string[] selectedCourses)
        {
            var newInstructor = new Instructor();

            if (selectedCourses.Length > 0)
            {
                newInstructor.Courses = new List<Course>();
                // Load collection with one DB call.
                _context.Courses.Load();
            }

            // Add selected Courses courses to the new instructor.
            foreach (var course in selectedCourses)
            {
                var foundCourse = await _context.Courses.FindAsync(int.Parse(course));
                if (foundCourse != null)
                {
                    newInstructor.Courses.Add(foundCourse);
                }
                else
                {
                    _logger.LogWarning("Course {course} not found", course);
                }
            }

            try
```

```
            {
                if (await TryUpdateModelAsync<Instructor>(
                                newInstructor,
                                "Instructor",
                                i => i.FirstMidName, i => i.LastName,
                                i => i.HireDate, i => i.OfficeAssignment))
                {
                    _context.Instructors.Add(newInstructor);
                    await _context.SaveChangesAsync();
                    return RedirectToPage("./Index");
                }
                return RedirectToPage("./Index");
            }
            catch (Exception ex)
            {
                _logger.LogError(ex.Message);
            }

            PopulateAssignedCourseData(_context, newInstructor);
            return Page();
        }
    }
}
```

The preceding code:

- Adds logging for warning and error messages.
- Calls Load, which fetches all the Courses in one database call. For small collections this is an optimization when using FindAsync. `FindAsync` returns the tracked entity without a request to the database.

```
public async Task<IActionResult> OnPostAsync(string[] selectedCourses)
{
    var newInstructor = new Instructor();

    if (selectedCourses.Length > 0)
    {
        newInstructor.Courses = new List<Course>();
        // Load collection with one DB call.
        _context.Courses.Load();
    }

    // Add selected Courses courses to the new instructor.
    foreach (var course in selectedCourses)
    {
        var foundCourse = await _context.Courses.FindAsync(int.Parse(course));
        if (foundCourse != null)
        {
            newInstructor.Courses.Add(foundCourse);
        }
        else
        {
            _logger.LogWarning("Course {course} not found", course);
        }
    }

    try
    {
        if (await TryUpdateModelAsync<Instructor>(
                        newInstructor,
                        "Instructor",
                        i => i.FirstMidName, i => i.LastName,
                        i => i.HireDate, i => i.OfficeAssignment))
        {
            _context.Instructors.Add(newInstructor);
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
        return RedirectToPage("./Index");
    }
    catch (Exception ex)
    {
        _logger.LogError(ex.Message);
    }

    PopulateAssignedCourseData(_context, newInstructor);
    return Page();
}
```

- `_context.Instructors.Add(newInstructor)` creates a new `Instructor` using many-to-many relationships without explicitly mapping the join table. Many-to-many was added in EF 5.0.

Test the instructor Create page.

Update the Instructor Create Razor page with code similar to the Edit page:

```
@page
@model ContosoUniversity.Pages.Instructors.CreateModel

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

<h2>Create</h2>

<h4>Instructor</h4>
```

```
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="table">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    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>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Update the Instructor Delete page

Update `Pages/Instructors/Delete.cshtml.cs` with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

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

            Instructor = await _context.Instructors.FirstOrDefaultAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            return Page();
        }

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

            Instructor instructor = await _context.Instructors
                .Include(i => i.Courses)
                .SingleAsync(i => i.ID == id);

            if (instructor == null)
            {
                return RedirectToPage("./Index");
            }

            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 RedirectToPage("./Index");
        }
    }
}
```

The preceding code makes the following changes:

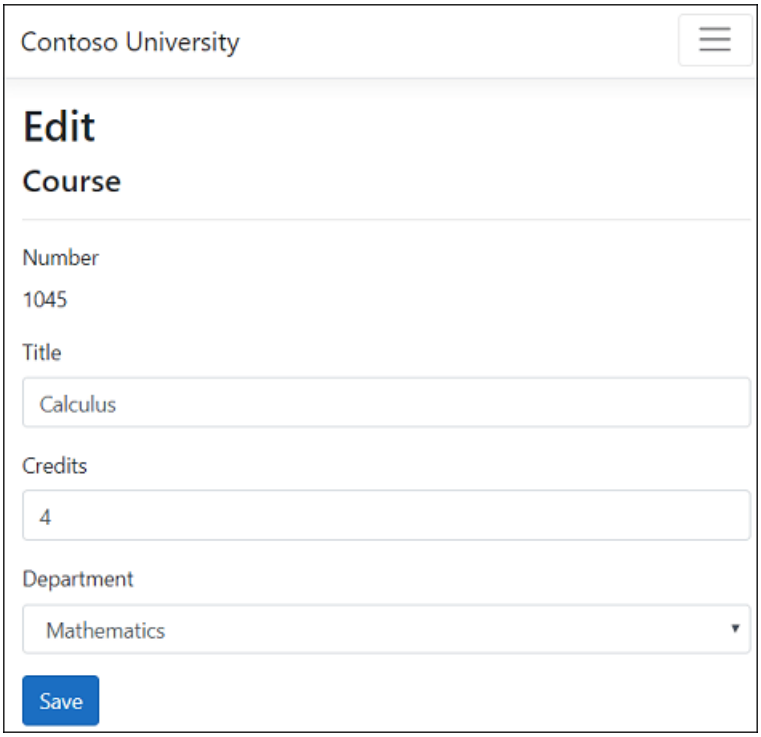
- Uses eager loading for the `Courses` navigation property. `Courses` must be included or they aren't deleted when the instructor is deleted. To avoid needing to read them, configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

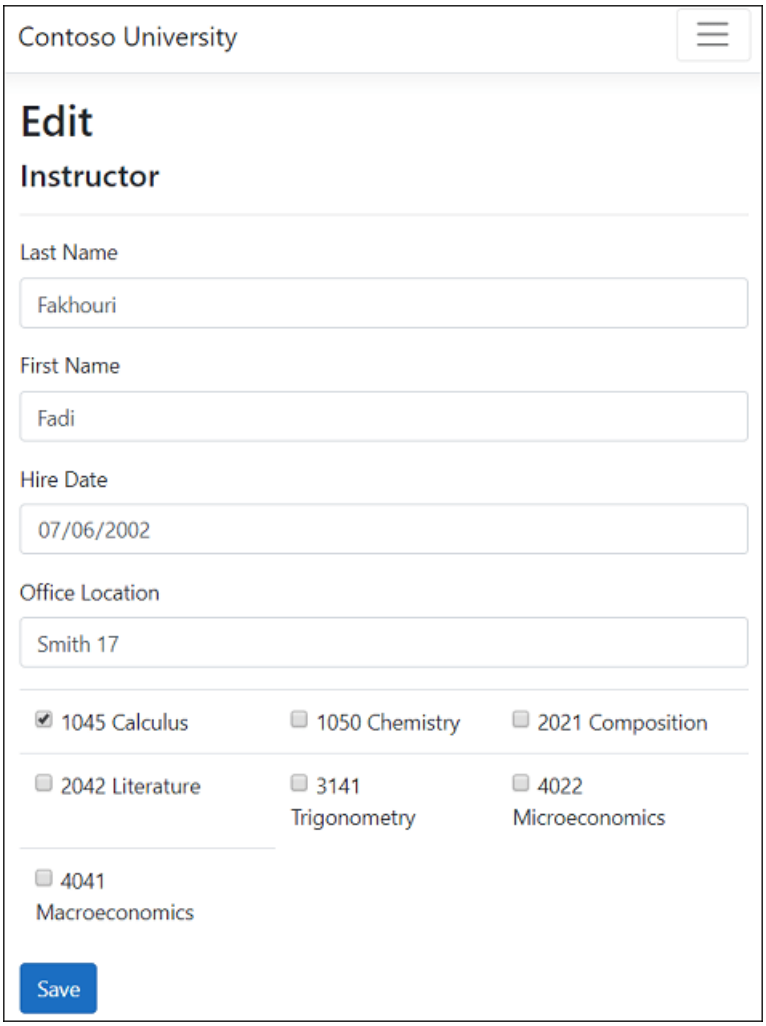
Run the app and test the Delete page.

Next steps

PREVIOUS TUTORIAL | NEXT TUTORIAL

This tutorial shows how to update related data. The following illustrations show some of the completed pages.





Update the Course Create and Edit pages

The scaffolded code for the Course Create and Edit pages has a Department drop-down list that shows Department ID (an integer). The drop-down should show the Department name, so both of these pages need a list of department names. To provide that list, use a base class for the Create and Edit pages.

Create a base class for Course Create and Edit

Create a `Pages/Courses/DepartmentNamePageModel.cs` file with the following code:

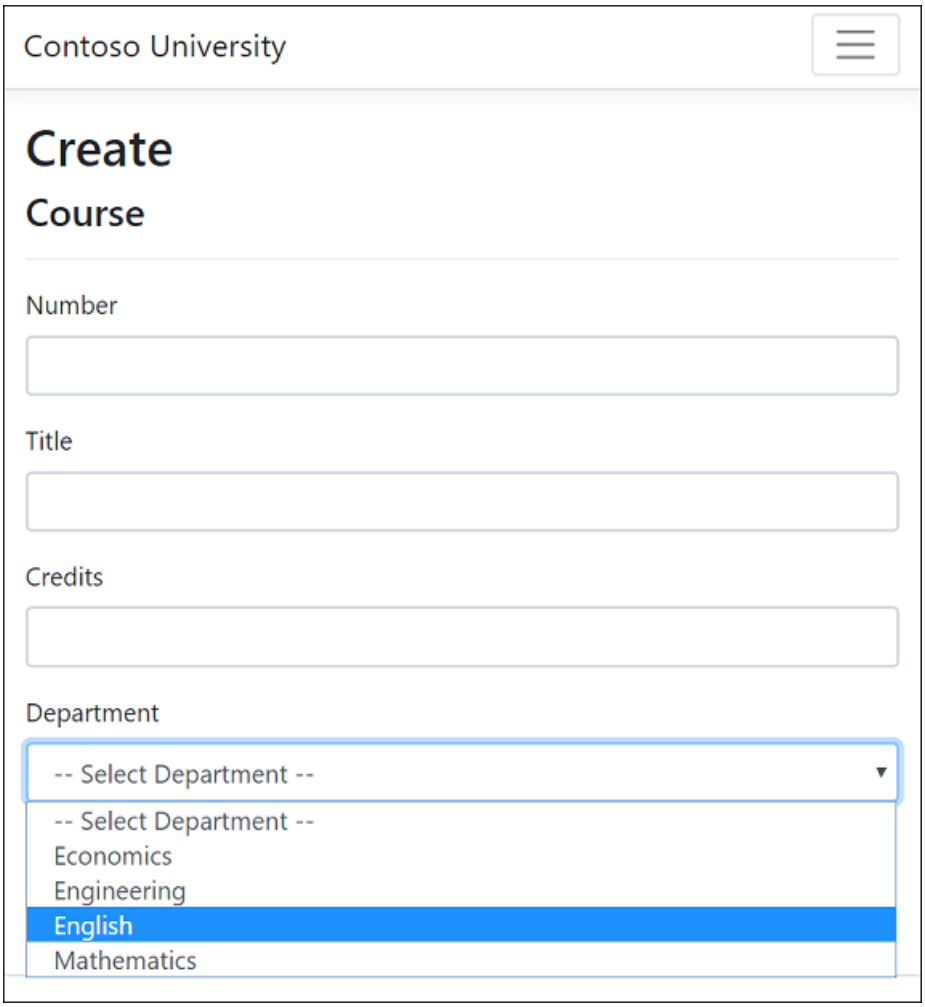
```
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace ContosoUniversity.Pages.Courses
{
    public class DepartmentNamePageModel : PageModel
    {
        public SelectList DepartmentNameSL { get; set; }

        public void PopulateDepartmentsDropDownList(SchoolContext _context,
            object selectedDepartment = null)
        {
            var departmentsQuery = from d in _context.Departments
                                   orderby d.Name // Sort by name.
                                   select d;

            DepartmentNameSL = new SelectList(departmentsQuery.AsNoTracking(),
                        "DepartmentID", "Name", selectedDepartment);
        }
    }
}
```

The preceding code creates a SelectList to contain the list of department names. If `selectedDepartment` is specified, that department is selected in the `SelectList`.

The Create and Edit page model classes will derive from `DepartmentNamePageModel`.

Update the Course Create page model

A Course is assigned to a Department. The base class for the Create and Edit pages provides a `SelectList` for selecting the department. The drop-down list that uses the `SelectList` sets the `Course.DepartmentID` foreign key (FK) property. EF Core uses the `Course.DepartmentID` FK to load the `Department` navigation property.



Update `Pages/Courses/Create.cshtml.cs` with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class CreateModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            PopulateDepartmentsDropDownList(_context);
            return Page();
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            var emptyCourse = new Course();

            if (await TryUpdateModelAsync<Course>(
                 emptyCourse,
                 "course",   // Prefix for form value.
                 s => s.CourseID, s => s.DepartmentID, s => s.Title, s => s.Credits))
            {
                _context.Courses.Add(emptyCourse);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, emptyCourse.DepartmentID);
            return Page();
        }
      }
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

The preceding code:

- Derives from `DepartmentNamePageModel`.
- Uses `TryUpdateModelAsync` to prevent overposting.
- Removes `ViewData["DepartmentID"]`. `DepartmentNameSL` from the base class is a strongly typed model and will be used by the Razor page. Strongly typed models are preferred over weakly typed. For more information, see Weakly typed data (ViewData and ViewBag).

Update the Course Create Razor page

Update `Pages/Courses/Create.cshtml` with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.CreateModel
@{
    ViewData["Title"] = "Create Course";
}
<h2>Create</h2>
<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <input asp-for="Course.CourseID" class="form-control" />
                <span asp-validation-for="Course.CourseID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL">
                    <option value="">-- Select Department --</option>
                </select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code makes the following changes:

- Changes the caption from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).
- Adds the "Select Department" option. This change renders "Select Department" in the drop-down when no department has been selected yet, rather than the first department.
- Adds a validation message when the department isn't selected.

The Razor Page uses the Select Tag Helper:

```
<div class="form-group">
    <label asp-for="Course.Department" class="control-label"></label>
    <select asp-for="Course.DepartmentID" class="form-control"
            asp-items="@Model.DepartmentNameSL">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="Course.DepartmentID" class="text-danger" />
</div>
```

Test the Create page. The Create page displays the department name rather than the department ID.

Update the Course Edit page model

Update `Pages/Courses/Edit.cshtml.cs` with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class EditModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

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

            Course = await _context.Courses
                .Include(c => c.Department).FirstOrDefaultAsync(m => m.CourseID == id);

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

            // Select current DepartmentID.
            PopulateDepartmentsDropDownList(_context, Course.DepartmentID);
            return Page();
        }

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

            var courseToUpdate = await _context.Courses.FindAsync(id);

            if (courseToUpdate == null)
            {
                return NotFound();
```

```
            }

            if (await TryUpdateModelAsync<Course>(
                 courseToUpdate,
                 "course",   // Prefix for form value.
                   c => c.Credits, c => c.DepartmentID, c => c.Title))
            {
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, courseToUpdate.DepartmentID);
            return Page();
        }
    }
}
```

The changes are similar to those made in the Create page model. In the preceding code, `PopulateDepartmentsDropDownList` passes in the department ID, which selects that department in the drop-down list.

Update the Course Edit Razor page

Update `Pages/Courses/Edit.cshtml` with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.EditModel

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

<h2>Edit</h2>

<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Course.CourseID" />
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <div>@Html.DisplayFor(model => model.Course.CourseID)</div>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL"></select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code makes the following changes:

- Displays the course ID. Generally the Primary Key (PK) of an entity isn't displayed. PKs are usually meaningless to users. In this case, the PK is the course number.
- Changes the caption for the Department drop-down from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).

The page contains a hidden field (`<input type="hidden">`) for the course number. Adding a `<label>` tag helper with `asp-for="Course.CourseID"` doesn't eliminate the need for the hidden field. `<input type="hidden">` is required for the course number to be included in the posted data when the user clicks **Save**.

Update the Course Details and Delete pages

AsNoTracking can improve performance when tracking isn't required.

Update the Course page models

Update `Pages/Courses/Delete.cshtml.cs` with the following code to add `AsNoTracking`:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

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

            Course = await _context.Courses
                .AsNoTracking()
                .Include(c => c.Department)
                .FirstOrDefaultAsync(m => m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }
            return Page();
        }

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

            Course = await _context.Courses.FindAsync(id);

            if (Course != null)
            {
                _context.Courses.Remove(Course);
                await _context.SaveChangesAsync();
            }

            return RedirectToPage("./Index");
        }
    }
}
```

Make the same change in the `Pages/Courses/Details.cshtml.cs` file:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class DetailsModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DetailsModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public Course Course { get; set; }

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

            Course = await _context.Courses
                 .AsNoTracking()
                 .Include(c => c.Department)
                 .FirstOrDefaultAsync(m => m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }
            return Page();
        }
    }
}
```

Update the Course Razor pages

Update `Pages/Courses/Delete.cshtml` with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.DeleteModel

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

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Department.Name)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Course.CourseID" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Make the same changes to the Details page.

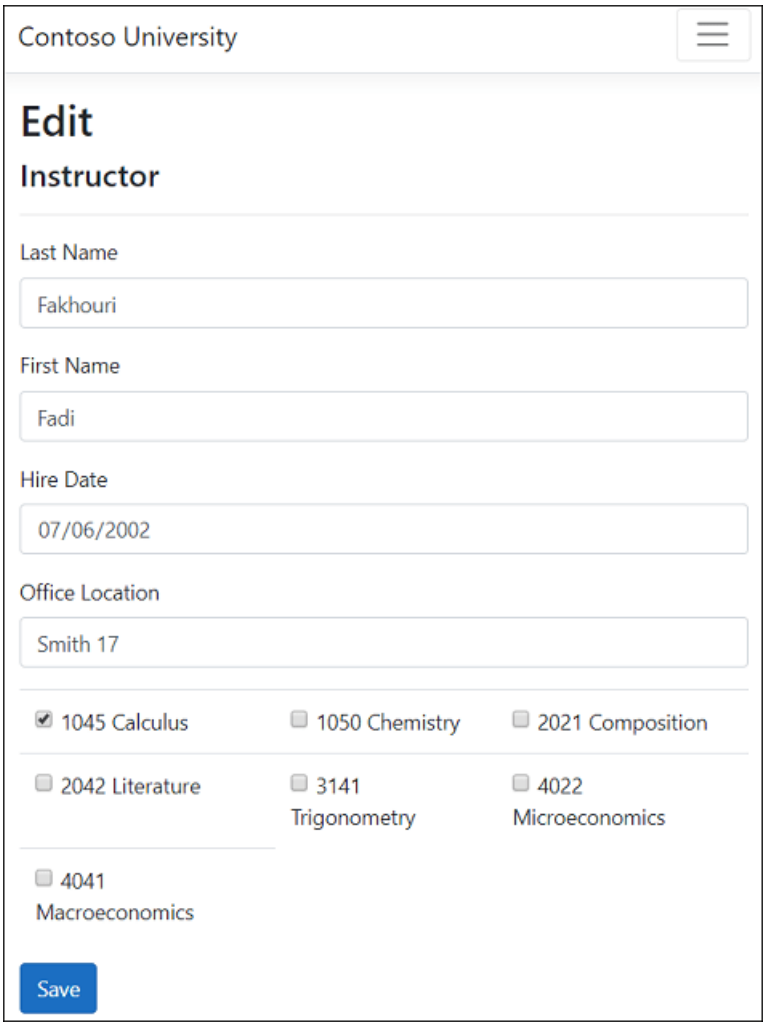
```
@page
@model ContosoUniversity.Pages.Courses.DetailsModel

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

<h2>Details</h2>

<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Department.Name)
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Course.CourseID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

Test the Course pages

Test the create, edit, details, and delete pages.

Update the instructor Create and Edit pages

Instructors may teach any number of courses. The following image shows the instructor Edit page with an array of course checkboxes.



The checkboxes enable changes to courses an instructor is assigned to. A checkbox is displayed for every course in the database. Courses that the instructor is assigned to are selected. The user can select or clear checkboxes to change course assignments. If the number of courses were much greater, a different UI might work better. But the method of managing a many-to-many relationship shown here wouldn't change. To create or delete relationships, you manipulate a join entity.

Create a class for assigned courses data

Create `Models/SchoolViewModels/AssignedCourseData.cs` with the following code:

```
namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

The `AssignedCourseData` class contains data to create the checkboxes for courses assigned to an instructor.

Create an Instructor page model base class

Create the `Pages/Instructors/InstructorCoursesPageModel.cs` base class:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
```

```
using System.Linq;

namespace ContosoUniversity.Pages.Instructors
{
    public class InstructorCoursesPageModel : PageModel
    {

        public List<AssignedCourseData> AssignedCourseDataList;

        public void PopulateAssignedCourseData(SchoolContext context,
                                               Instructor instructor)
        {
            var allCourses = context.Courses;
            var instructorCourses = new HashSet<int>(
                instructor.CourseAssignments.Select(c => c.CourseID));
            AssignedCourseDataList = new List<AssignedCourseData>();
            foreach (var course in allCourses)
            {
                AssignedCourseDataList.Add(new AssignedCourseData
                {
                    CourseID = course.CourseID,
                    Title = course.Title,
                    Assigned = instructorCourses.Contains(course.CourseID)
                });
            }
        }

        public void UpdateInstructorCourses(SchoolContext context,
            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);
                    }
                }
            }
        }
    }
}
```

The `InstructorCoursesPageModel` is the base class you will use for the Edit and Create page models. `PopulateAssignedCourseData` reads all `Course` entities to populate `AssignedCourseDataList`. For each course, the code sets the `CourseID`, title, and whether or not the instructor is assigned to the course. A HashSet is used for efficient lookups.

Since the Razor page doesn't have a collection of Course entities, the model binder can't automatically update the `CourseAssignments` navigation property. Instead of using the model binder to update the `CourseAssignments` navigation property, you do that in the new `UpdateInstructorCourses` method. Therefore you need to exclude the `CourseAssignments` property from model binding. This doesn't require any change to the code that calls `TryUpdateModel` because you're using the overload with declared properties and `CourseAssignments` isn't in the include list.

If no checkboxes were selected, the code in `UpdateInstructorCourses` initializes the `CourseAssignments` navigation property with an empty collection and returns:

```
if (selectedCourses == null)
{
    instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
    return;
}
```

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 page. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the checkbox 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.

```
if (selectedCoursesHS.Contains(course.CourseID.ToString()))
{
    if (!instructorCourses.Contains(course.CourseID))
    {
        instructorToUpdate.CourseAssignments.Add(
            new CourseAssignment
            {
                InstructorID = instructorToUpdate.ID,
                CourseID = course.CourseID
            });
    }
}
```

If the checkbox for a course wasn't selected, but the course is in the `Instructor.CourseAssignments` navigation property, the course is removed from the navigation property.

```
else
{
    if (instructorCourses.Contains(course.CourseID))
    {
        CourseAssignment courseToRemove
            = instructorToUpdate
                .CourseAssignments
                .SingleOrDefault(i => i.CourseID == course.CourseID);
        context.Remove(courseToRemove);
    }
}
```

Handle office location

Another relationship the edit page has to handle is the one-to-zero-or-one relationship that the Instructor entity has with the `OfficeAssignment` entity. The instructor edit code must handle the following scenarios:

- If the user clears the office assignment, delete the `OfficeAssignment` entity.
- If the user enters an office assignment and it was empty, create a new `OfficeAssignment` entity.
- If the user changes the office assignment, update the `OfficeAssignment` entity.

Update the Instructor Edit page model

Update `Pages/Instructors/Edit.cshtml.cs` with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class EditModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

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

            Instructor = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            PopulateAssignedCourseData(_context, Instructor);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(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)
                .FirstOrDefaultAsync(s => s.ID == id);

            if (instructorToUpdate == null)
            {
```

```
            {
                return NotFound();
            }

            if (await TryUpdateModelAsync<Instructor>(
                instructorToUpdate,
                "Instructor",
                i => i.FirstMidName, i => i.LastName,
                i => i.HireDate, i => i.OfficeAssignment))
            {
                if (String.IsNullOrWhiteSpace(
                    instructorToUpdate.OfficeAssignment?.Location))
                {
                    instructorToUpdate.OfficeAssignment = null;
                }
                UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
            PopulateAssignedCourseData(_context, instructorToUpdate);
            return Page();
        }
    }
}
```

The preceding code:

- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment`, `CourseAssignment`, and `CourseAssignment.Course` navigation properties.
- Updates the retrieved `Instructor` entity with values from the model binder. `TryUpdateModel` prevents overposting.
- If the office location is blank, sets `Instructor.OfficeAssignment` to null. When `Instructor.OfficeAssignment` is null, the related row in the `OfficeAssignment` table is deleted.
- Calls `PopulateAssignedCourseData` in `OnGetAsync` to provide information for the checkboxes using the `AssignedCourseData` view model class.
- Calls `UpdateInstructorCourses` in `OnPostAsync` to apply information from the checkboxes to the Instructor entity being edited.
- Calls `PopulateAssignedCourseData` and `UpdateInstructorCourses` in `OnPostAsync` if `TryUpdateModel` fails. These method calls restore the assigned course data entered on the page when it is redisplayed with an error message.

Update the Instructor Edit Razor page

Update `Pages/Instructors/Edit.cshtml` with the following code:

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
```

```
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="table">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    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>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code creates an HTML table that has three columns. Each column has a checkbox and a caption containing the course number and title. The checkboxes all have the same name ("selectedCourses"). Using the same name informs the model binder to treat them as a group. The value attribute of each checkbox is set to `CourseID`. When the page is posted, the model binder passes an array that consists of the `CourseID` values for only the checkboxes that are selected.

When the checkboxes are initially rendered, courses assigned to the instructor are selected.

Note: The approach taken here to edit instructor course data works well when there's a limited number of

courses. For collections that are much larger, a different UI and a different updating method would be more useable and efficient.

Run the app and test the updated Instructors Edit page. Change some course assignments. The changes are reflected on the Index page.

Update the Instructor Create page

Update the Instructor Create page model and Razor page with code similar to the Edit page:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class CreateModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            var instructor = new Instructor();
            instructor.CourseAssignments = new List<CourseAssignment>();

            // Provides an empty collection for the foreach loop
            // foreach (var course in Model.AssignedCourseDataList)
            // in the Create Razor page.
            PopulateAssignedCourseData(_context, instructor);
            return Page();
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnPostAsync(string[] selectedCourses)
        {
            var newInstructor = new Instructor();
            if (selectedCourses != null)
            {
                newInstructor.CourseAssignments = new List<CourseAssignment>();
                foreach (var course in selectedCourses)
                {
                    var courseToAdd = new CourseAssignment
                    {
                        CourseID = int.Parse(course)
                    };
                    newInstructor.CourseAssignments.Add(courseToAdd);
                }
            }

            if (await TryUpdateModelAsync<Instructor>(
                newInstructor,
                "Instructor",
                i => i.FirstMidName, i => i.LastName,
                i => i.HireDate, i => i.OfficeAssignment))
            {
                _context.Instructors.Add(newInstructor);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            PopulateAssignedCourseData(_context, newInstructor);
            return Page();
        }
    }
}
```

```
@page
@model ContosoUniversity.Pages.Instructors.CreateModel
```

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

<h2>Create</h2>

<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="table">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    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>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>
```

```
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Test the instructor Create page.

Update the Instructor Delete page

Update `Pages/Instructors/Delete.cshtml.cs` with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

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

            Instructor = await _context.Instructors.FirstOrDefaultAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            return Page();
        }

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

            Instructor instructor = await _context.Instructors
                .Include(i => i.CourseAssignments)
                .SingleAsync(i => i.ID == id);

            if (instructor == null)
            {
                return RedirectToPage("./Index");
            }

            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 RedirectToPage("./Index");
        }
    }
}
```

The preceding code makes the following changes:

- Uses eager loading for the `CourseAssignments` navigation property. `CourseAssignments` must be included or they aren't deleted when the instructor is deleted. To avoid needing to read them, 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.

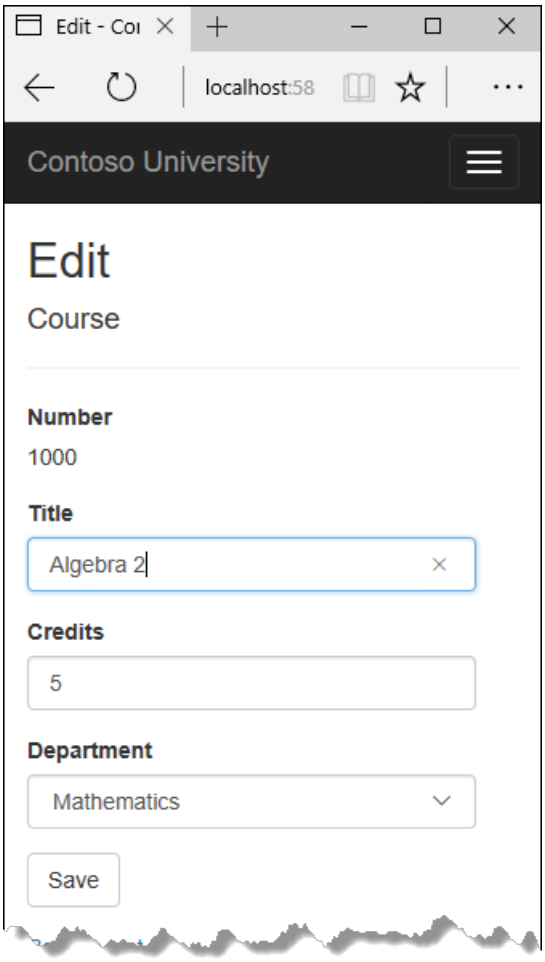
Run the app and test the Delete page.

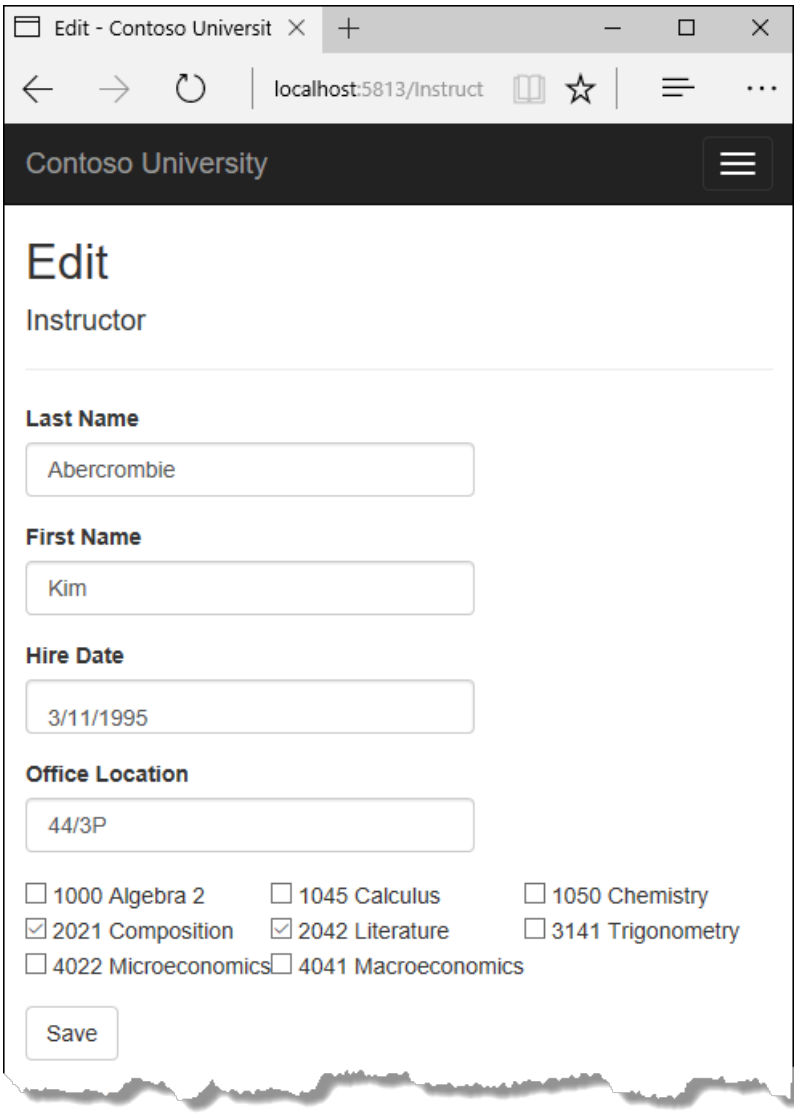
Next steps

PREVIOUS TUTORIAL | NEXT TUTORIAL

This tutorial demonstrates updating related data. If you run into problems you can't solve, download or view the completed app. Download instructions.

The following illustrations shows some of the completed pages.





Examine and test the Create and Edit course pages. Create a new course. The department is selected by its primary key (an integer), not its name. Edit the new course. When you have finished testing, delete the new course.

Create a base class to share common code

The Courses/Create and Courses/Edit pages each need a list of department names. Create the `Pages/Courses/DepartmentNamePageModel.cshtml.cs` base class for the Create and Edit pages:

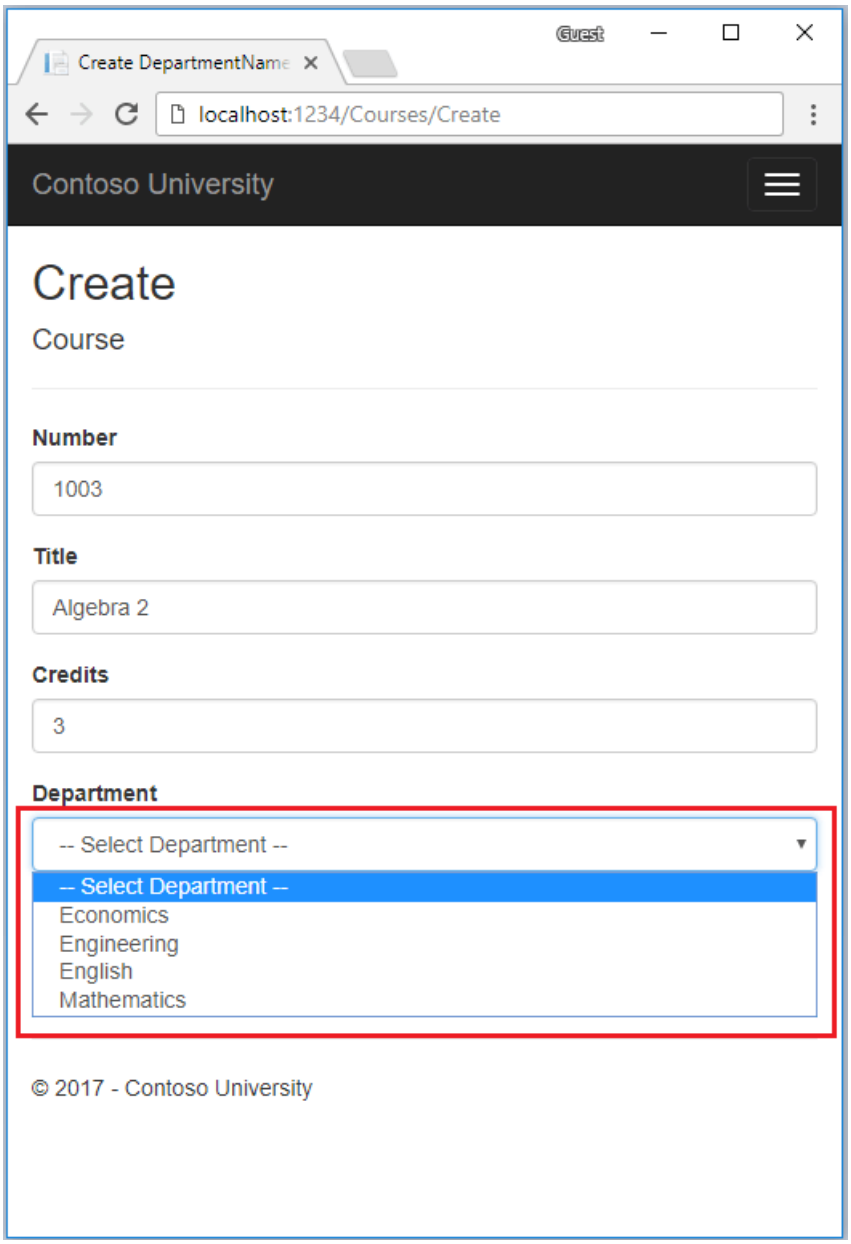
```
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace ContosoUniversity.Pages.Courses
{
    public class DepartmentNamePageModel : PageModel
    {
        public SelectList DepartmentNameSL { get; set; }

        public void PopulateDepartmentsDropDownList(SchoolContext _context,
            object selectedDepartment = null)
        {
            var departmentsQuery = from d in _context.Departments
                                   orderby d.Name // Sort by name.
                                   select d;

            DepartmentNameSL = new SelectList(departmentsQuery.AsNoTracking(),
                        "DepartmentID", "Name", selectedDepartment);
        }
    }
}
```

The preceding code creates a SelectList to contain the list of department names. If `selectedDepartment` is specified, that department is selected in the `SelectList`.

The Create and Edit page model classes will derive from `DepartmentNamePageModel`.

Customize the Courses Pages

When a new course entity is created, it must have a relationship to an existing department. To add a department while creating a course, the base class for Create and Edit contains a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key (FK) property. EF Core uses the `Course.DepartmentID` FK to load the `Department` navigation property.



Update the Create page model with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class CreateModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            PopulateDepartmentsDropDownList(_context);
            return Page();
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var emptyCourse = new Course();

            if (await TryUpdateModelAsync<Course>(
                 emptyCourse,
                 "course",   // Prefix for form value.
                 s => s.CourseID, s => s.DepartmentID, s => s.Title, s => s.Credits))
            {
                _context.Courses.Add(emptyCourse);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, emptyCourse.DepartmentID);
            return Page();
        }
      }
}
```

The preceding code:

- Derives from `DepartmentNamePageModel`.
- Uses `TryUpdateModelAsync` to prevent overposting.
- Replaces `ViewData["DepartmentID"]` with `DepartmentNameSL` (from the base class).

`ViewData["DepartmentID"]` is replaced with the strongly typed `DepartmentNameSL`. Strongly typed models are preferred over weakly typed. For more information, see Weakly typed data (ViewData and ViewBag).

Update the Courses Create page

Update `Pages/Courses/Create.cshtml` with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.CreateModel
@{
    ViewData["Title"] = "Create Course";
}
<h2>Create</h2>
<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <input asp-for="Course.CourseID" class="form-control" />
                <span asp-validation-for="Course.CourseID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL">
                    <option value="">-- Select Department --</option>
                </select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding markup makes the following changes:

- Changes the caption from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).
- Adds the "Select Department" option. This change renders "Select Department" rather than the first department.
- Adds a validation message when the department isn't selected.

The Razor Page uses the Select Tag Helper:

```
<div class="form-group">
    <label asp-for="Course.Department" class="control-label"></label>
    <select asp-for="Course.DepartmentID" class="form-control"
            asp-items="@Model.DepartmentNameSL">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="Course.DepartmentID" class="text-danger" />
</div>
```

Test the Create page. The Create page displays the department name rather than the department ID.

Update the Courses Edit page.

Replace the code in `Pages/Courses/Edit.cshtml.cs` with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class EditModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

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

            Course = await _context.Courses
                .Include(c => c.Department).FirstOrDefaultAsync(m => m.CourseID == id);

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

            // Select current DepartmentID.
            PopulateDepartmentsDropDownList(_context,Course.DepartmentID);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var courseToUpdate = await _context.Courses.FindAsync(id);

            if (await TryUpdateModelAsync<Course>(
                 courseToUpdate,
                 "course",   // Prefix for form value.
                   c => c.Credits, c => c.DepartmentID, c => c.Title))
            {
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, courseToUpdate.DepartmentID);
            return Page();
        }
    }
}
```

The changes are similar to those made in the Create page model. In the preceding code, `PopulateDepartmentsDropDownList` passes in the department ID, which select the department specified in the

drop-down list.

Update `Pages/Courses/Edit.cshtml` with the following markup:

```
@page
@model ContosoUniversity.Pages.Courses.EditModel

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

<h2>Edit</h2>

<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Course.CourseID" />
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <div>@Html.DisplayFor(model => model.Course.CourseID)</div>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL"></select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding markup makes the following changes:

- Displays the course ID. Generally the Primary Key (PK) of an entity isn't displayed. PKs are usually meaningless to users. In this case, the PK is the course number.
- Changes the caption from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).

The page contains a hidden field (`<input type="hidden">`) for the course number. Adding a `<label>` tag helper with `asp-for="Course.CourseID"` doesn't eliminate the need for the hidden field. `<input type="hidden">` is

required for the course number to be included in the posted data when the user clicks **Save**.

Test the updated code. Create, edit, and delete a course.

Add AsNoTracking to the Details and Delete page models

AsNoTracking can improve performance when tracking isn't required. Add `AsNoTracking` to the Delete and Details page model. The following code shows the updated Delete page model:

```
public class DeleteModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public DeleteModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Course Course { get; set; }

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

        Course = await _context.Courses
            .AsNoTracking()
            .Include(c => c.Department)
            .FirstOrDefaultAsync(m => m.CourseID == id);

        if (Course == null)
        {
            return NotFound();
        }
        return Page();
    }

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

        Course = await _context.Courses
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.CourseID == id);

        if (Course != null)
        {
            _context.Courses.Remove(Course);
            await _context.SaveChangesAsync();
        }

        return RedirectToPage("./Index");
    }
}
```

Update the `OnGetAsync` method in the `Pages/Courses/Details.cshtml.cs` file:

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

    Course = await _context.Courses
         .AsNoTracking()
         .Include(c => c.Department)
         .FirstOrDefaultAsync(m => m.CourseID == id);

    if (Course == null)
    {
        return NotFound();
    }
    return Page();
}
```

Modify the Delete and Details pages

Update the Delete Razor page with the following markup:

```
@page
@model ContosoUniversity.Pages.Courses.DeleteModel

@{
    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.Course.CourseID)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Department.DepartmentID)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Course.CourseID" />
        <input type="submit" value="Delete" class="btn btn-default" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Make the same changes to the Details page.

Test the Course pages

Test create, edit, details, and delete.

Update the instructor pages

The following sections update the instructor pages.

Add office location

When editing an instructor record, you may want to update the instructor's office assignment. The `Instructor` entity has a one-to-zero-or-one relationship with the `OfficeAssignment` entity. The instructor code must handle:

- If the user clears the office assignment, delete the `OfficeAssignment` entity.
- If the user enters an office assignment and it was empty, create a new `OfficeAssignment` entity.

- If the user changes the office assignment, update the `OfficeAssignment` entity.

Update the instructors Edit page model with the following code:

```
public class EditModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Instructor Instructor { get; set; }

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

        Instructor = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.ID == id);

        if (Instructor == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var instructorToUpdate = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .FirstOrDefaultAsync(s => s.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            if (String.IsNullOrWhiteSpace(
                instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            await _context.SaveChangesAsync();
        }
        return RedirectToPage("./Index");

    }
}
```

The preceding code:

- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment` navigation property.
- Updates the retrieved `Instructor` entity with values from the model binder. `TryUpdateModel` prevents overposting.
- If the office location is blank, sets `Instructor.OfficeAssignment` to null. When `Instructor.OfficeAssignment` is null, the related row in the `OfficeAssignment` table is deleted.

Update the instructor Edit page

Update `Pages/Instructors/Edit.cshtml` with the office location:

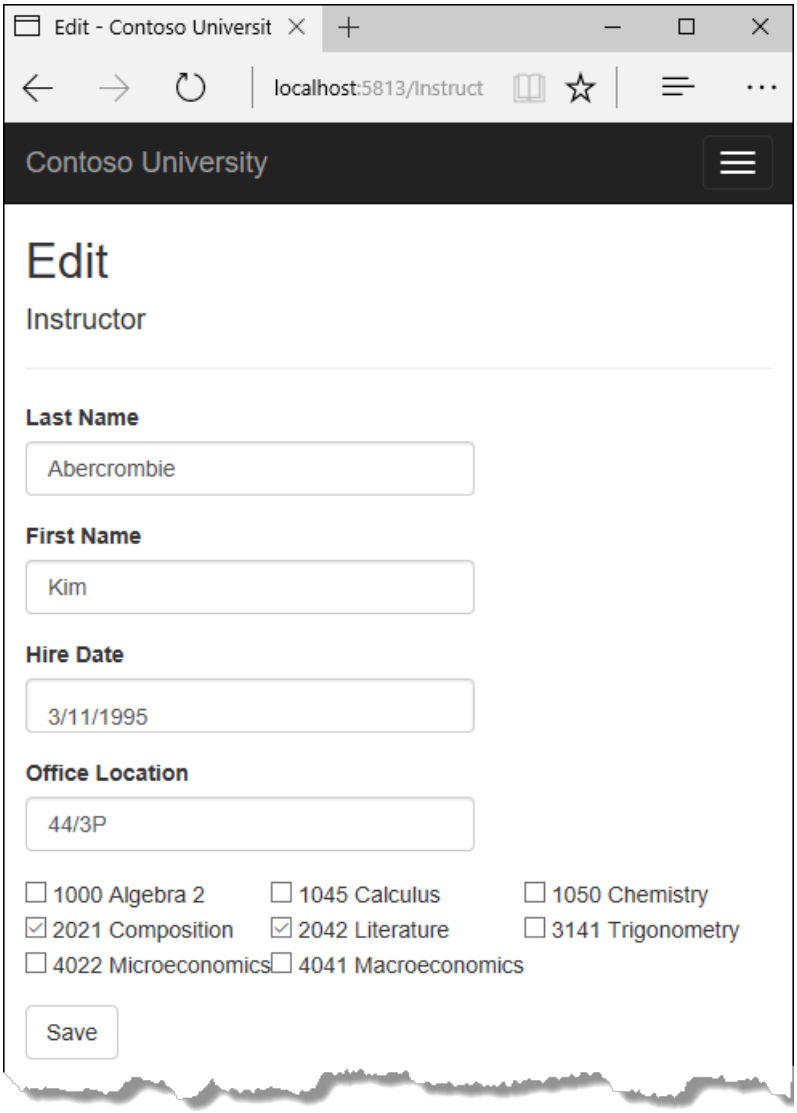
```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Verify you can change an instructors office location.

Add Course assignments to the instructor Edit page

Instructors may teach any number of courses. In this section, you add the ability to change course assignments.

The following image shows the updated instructor Edit page:



`Course` and `Instructor` has a many-to-many relationship. To add and remove relationships, you add and remove entities from the `CourseAssignments` join entity set.

checkboxes enable changes to courses an instructor is assigned to. A checkbox is displayed for every course in the database. Courses that the instructor is assigned to are checked. The user can select or clear checkboxes to change course assignments. If the number of courses were much greater:

- You'd probably use a different user interface to display the courses.
- The method of manipulating a join entity to create or delete relationships wouldn't change.

Add classes to support Create and Edit instructor pages

Create `Models/SchoolViewModels/AssignedCourseData.cs` with the following code:

```
namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

The `AssignedCourseData` class contains data to create the checkboxes for assigned courses by an instructor.

Create the `Pages/Instructors/InstructorCoursesPageModel.cshtml.cs` base class:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using System.Linq;

namespace ContosoUniversity.Pages.Instructors
{
    public class InstructorCoursesPageModel : PageModel
    {

        public List<AssignedCourseData> AssignedCourseDataList;

        public void PopulateAssignedCourseData(SchoolContext context,
                                               Instructor instructor)
        {
            var allCourses = context.Courses;
            var instructorCourses = new HashSet<int>(
                instructor.CourseAssignments.Select(c => c.CourseID));
            AssignedCourseDataList = new List<AssignedCourseData>();
            foreach (var course in allCourses)
            {
                AssignedCourseDataList.Add(new AssignedCourseData
                {
                    CourseID = course.CourseID,
                    Title = course.Title,
                    Assigned = instructorCourses.Contains(course.CourseID)
                });
            }
        }

        public void UpdateInstructorCourses(SchoolContext context,
            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);
                    }
                }
            }
        }
    }
}
```

The `InstructorCoursesPageModel` is the base class you will use for the Edit and Create page models. `PopulateAssignedCourseData` reads all `Course` entities to populate `AssignedCourseDataList`. For each course, the code sets the `CourseID`, title, and whether or not the instructor is assigned to the course. A HashSet is used to create efficient lookups.

Instructors Edit page model

Update the instructor Edit page model with the following code:

```
public class EditModel : InstructorCoursesPageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Instructor Instructor { get; set; }

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

        Instructor = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.ID == id);

        if (Instructor == null)
        {
            return NotFound();
        }
        PopulateAssignedCourseData(_context, Instructor);
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id, string[] selectedCourses)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var instructorToUpdate = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
            .FirstOrDefaultAsync(s => s.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            if (String.IsNullOrWhiteSpace(
                instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
        UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
        PopulateAssignedCourseData(_context, instructorToUpdate);
        return Page();
    }
}
```

The preceding code handles office assignment changes.

Update the instructor Razor View:

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    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>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
```

```
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

> **NOTE**
> When you paste the code in Visual Studio, line breaks are changed in a way that breaks the code. Press Ctrl+Z one time to undo the automatic formatting. Ctrl+Z fixes 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. With the block of new code selected, press Tab three times to line up the new code with the existing code. Vote on or review the status of this bug with this link.

The preceding code creates an HTML table that has three columns. Each column has a checkbox and a caption containing the course number and title. The checkboxes all have the same name ("selectedCourses"). Using the same name informs the model binder to treat them as a group. The value attribute of each checkbox is set to `CourseID`. When the page is posted, the model binder passes an array that consists of the `CourseID` values for only the checkboxes that are selected.

When the checkboxes are initially rendered, courses assigned to the instructor have checked attributes.

Run the app and test the updated instructors Edit page. Change some course assignments. The changes are reflected on the Index page.

Note: The approach taken here to edit instructor course data works well when there's a limited number of courses. For collections that are much larger, a different UI and a different updating method would be more useable and efficient.

Update the instructors Create page

Update the instructor Create page model with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class CreateModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            var instructor = new Instructor();
            instructor.CourseAssignments = new List<CourseAssignment>();

            // Provides an empty collection for the foreach loop
            // foreach (var course in Model.AssignedCourseDataList)
            // in the Create Razor page.
```

```
            PopulateAssignedCourseData(_context, instructor);
            return Page();
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnPostAsync(string[] selectedCourses)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var newInstructor = new Instructor();
            if (selectedCourses != null)
            {
                newInstructor.CourseAssignments = new List<CourseAssignment>();
                foreach (var course in selectedCourses)
                {
                    var courseToAdd = new CourseAssignment
                    {
                        CourseID = int.Parse(course)
                    };
                    newInstructor.CourseAssignments.Add(courseToAdd);
                }
            }

            if (await TryUpdateModelAsync<Instructor>(
                newInstructor,
                "Instructor",
                i => i.FirstMidName, i => i.LastName,
                i => i.HireDate, i => i.OfficeAssignment))
            {
                _context.Instructors.Add(newInstructor);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            PopulateAssignedCourseData(_context, newInstructor);
            return Page();
        }
    }
}
```

The preceding code is similar to the `Pages/Instructors/Edit.cshtml.cs` code.

Update the instructor Create Razor page with the following markup:

```
@page
@model ContosoUniversity.Pages.Instructors.CreateModel

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

<h2>Create</h2>

<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
```

```
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>

            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    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>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Test the instructor Create page.

Update the Delete page

Update the Delete page model with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

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

            Instructor = await _context.Instructors.SingleAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            Instructor instructor = await _context.Instructors
                .Include(i => i.CourseAssignments)
                .SingleAsync(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 RedirectToPage("./Index");
        }
    }
}
```

The preceding code makes the following changes:

- Uses eager loading for the `CourseAssignments` navigation property. `CourseAssignments` must be included or they aren't deleted when the instructor is deleted. To avoid needing to read them, 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.

Additional resources

- YouTube version of this tutorial (Part 1)
- YouTube version of this tutorial (Part 2)

PREVIOUS NEXT