```
            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 departments = new Department[]
            {
                english,
                mathematics,
                engineering,
                economics
            };

            context.AddRange(departments);

            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 courses = new Course[]
            {
                chemistry,
                microeconomics,
                macroeconmics,
                calculus,
                trigonometry,
                composition,
                literature
            };

            context.AddRange(courses);

            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

- Delete the *Migrations* folder.
- In the **Package Manager Console** (PMC), run the following commands:

PowerShell

```
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

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

Part 6, Razor Pages with EF Core in ASP.NET Core - Read Related Data

Article • 04/10/2024

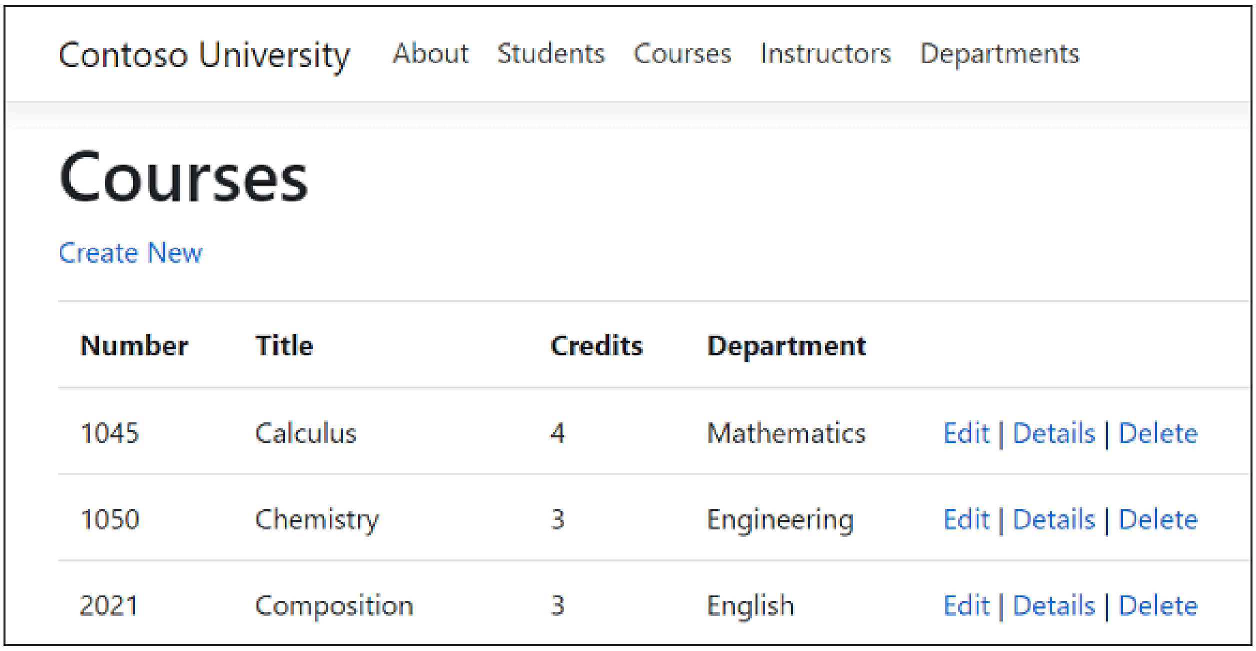
By Tom Dykstra, Jon P Smith, and Rick Anderson

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.

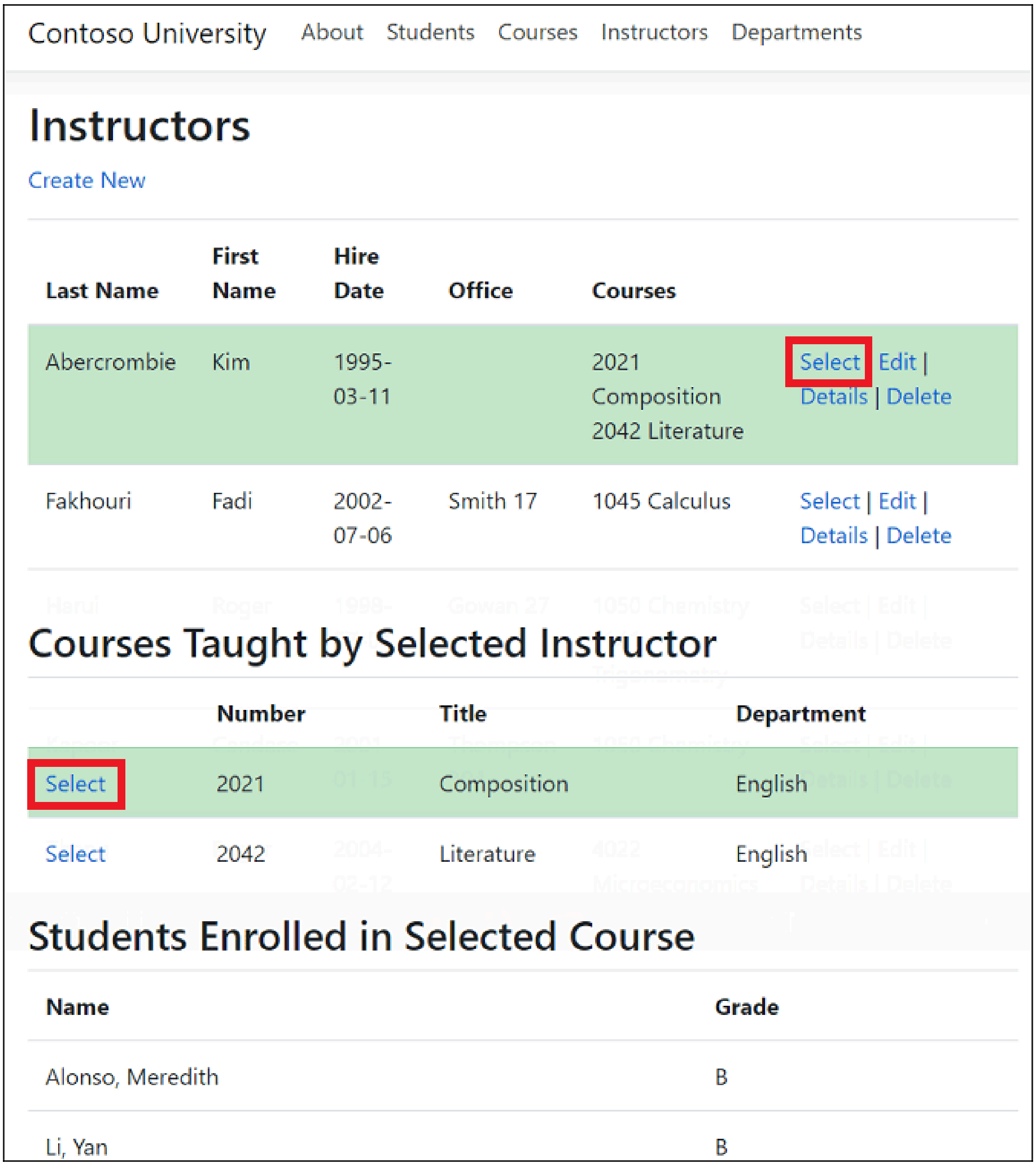
The following illustrations show the completed pages for this tutorial:



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



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

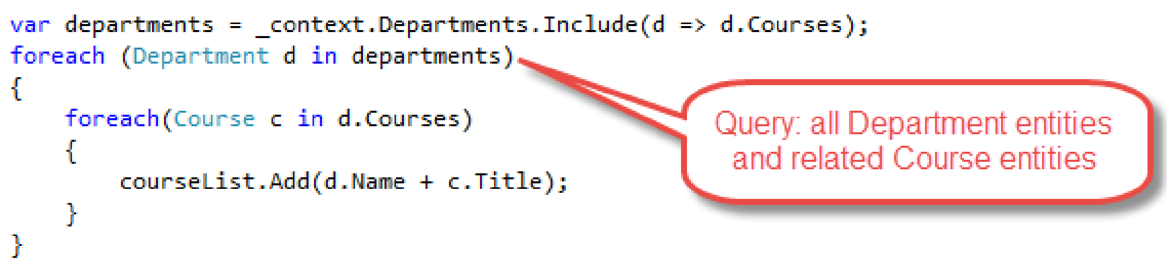
Students Enrolled in Selected Course

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 large single query. Eager loading is specified with the Include and ThenInclude methods.

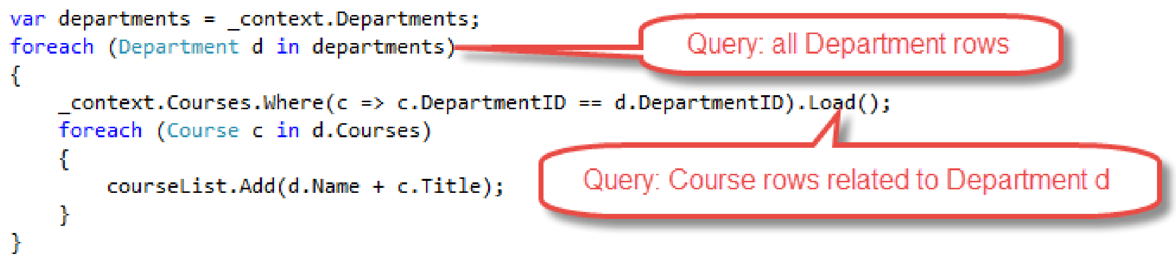


```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

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.

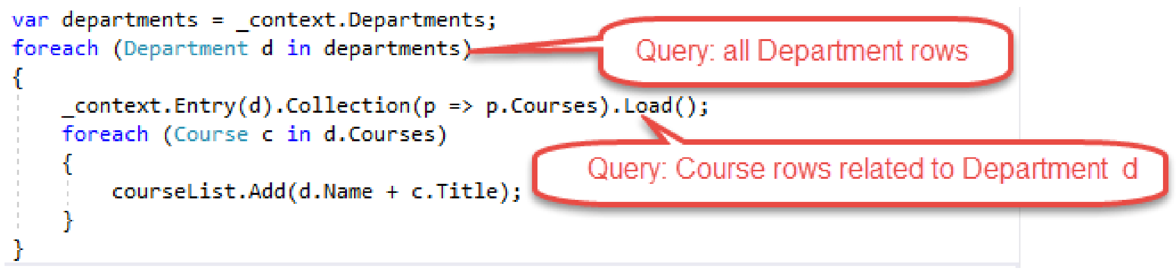


```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

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:



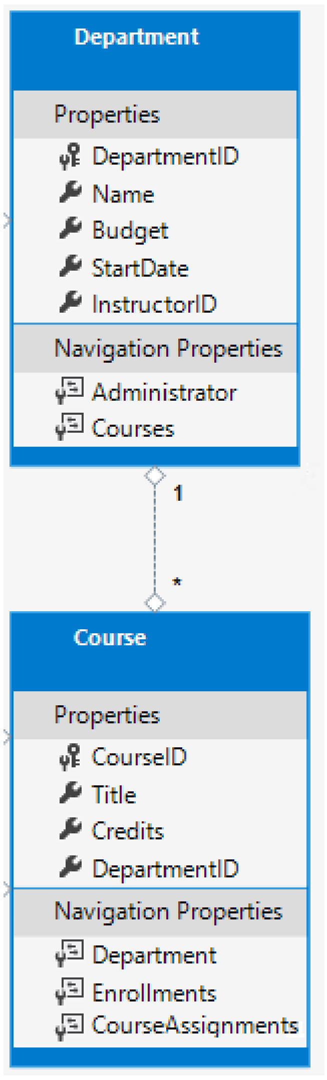
```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

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

- 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:

```C#
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`.

No-tracking queries are useful when the results are used in a read-only scenario. They're generally quicker to execute because there's no need to set up the change tracking information. If the entities retrieved from the database don't need to be updated, then a no-tracking query is likely to perform better than a tracking query.

In some cases a tracking query is more efficient than a no-tracking query. For more information, see Tracking vs. No-Tracking Queries. In the preceding code, `AsNoTracking` is called because the entities aren't updated in the current context.

Update `Pages/Courses/Index.cshtml` with the following code.

CSHTML

```
@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>
```

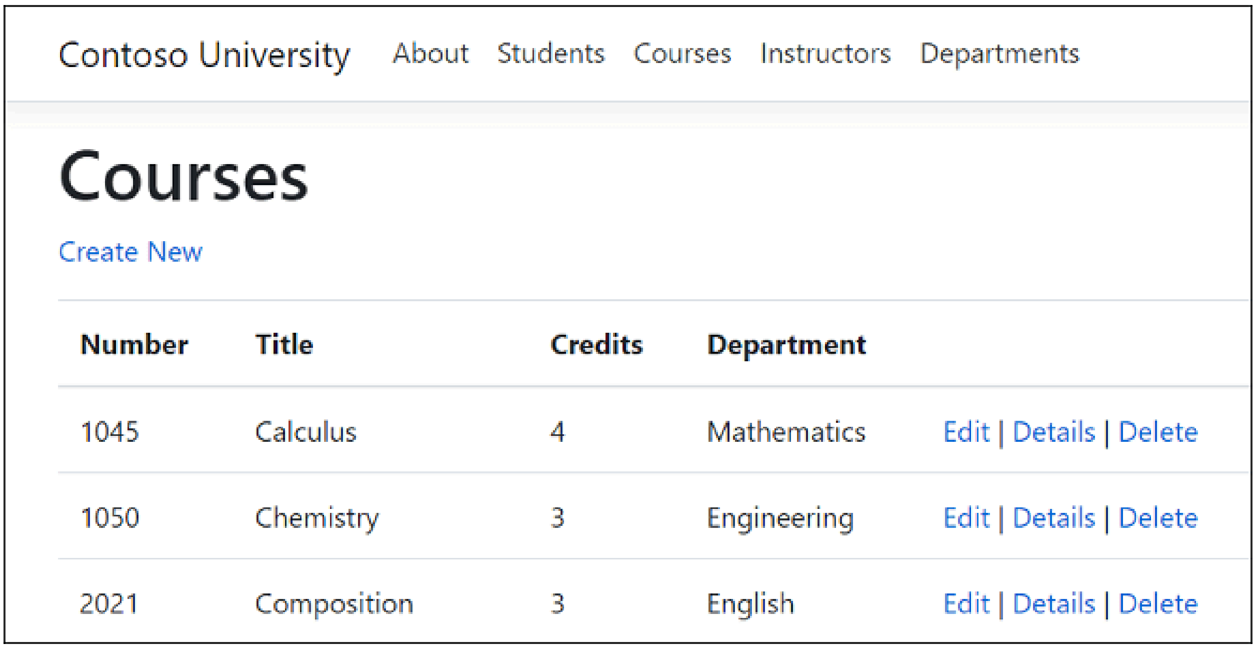
The following changes have been made to the scaffolded code:

- 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

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

Run the app and select the **Courses** tab to see the list with department names.



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

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:

C#

```csharp
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`:

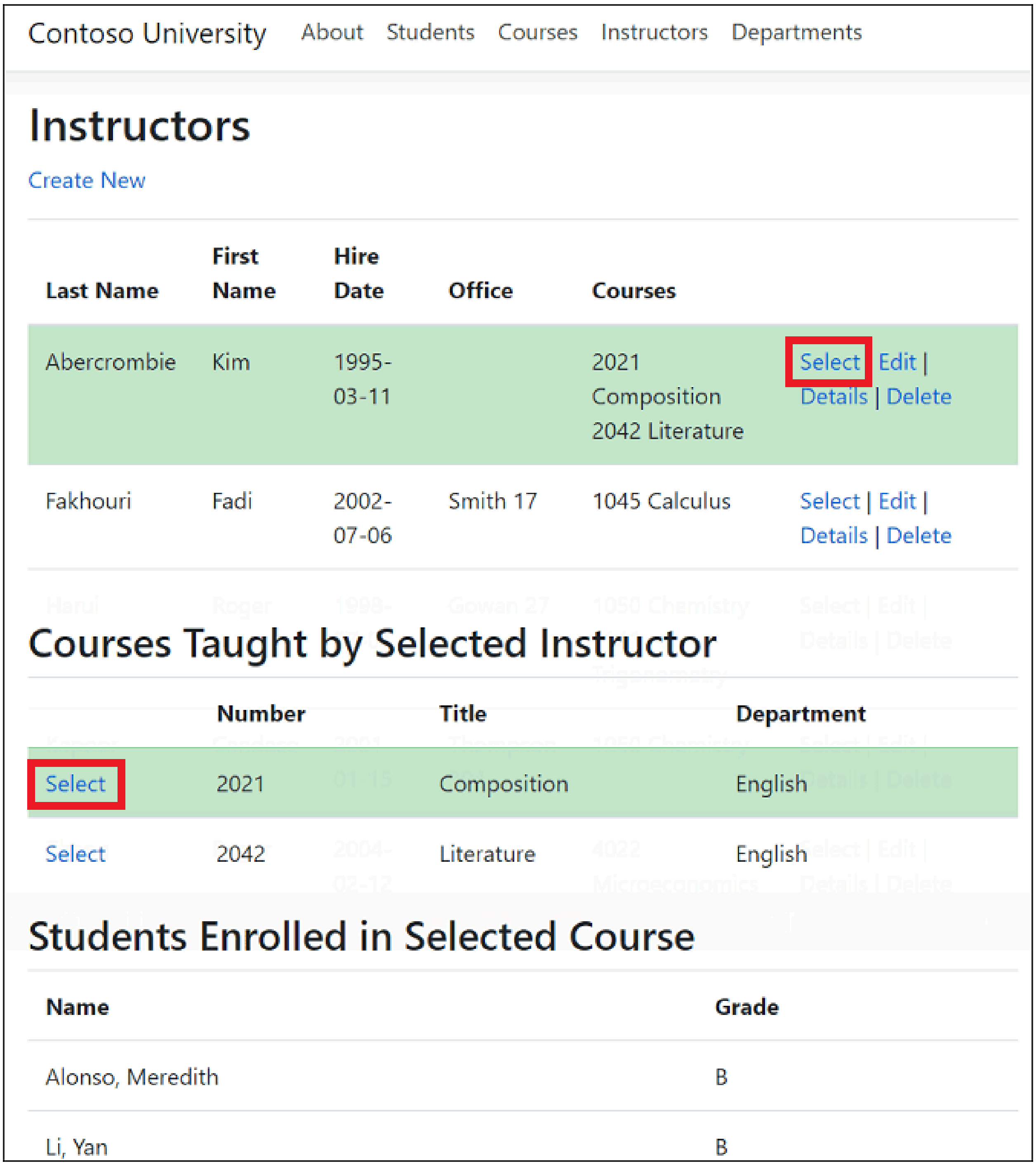
C#

```
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.



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

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:

C#

```
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

- 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:

C#

```
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:

```C#
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`.

```C#
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:

C#

```
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.

CSHTML

```
@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.

  HTML

  ```
  @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.

  HTML

  ```
  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.

  HTML

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

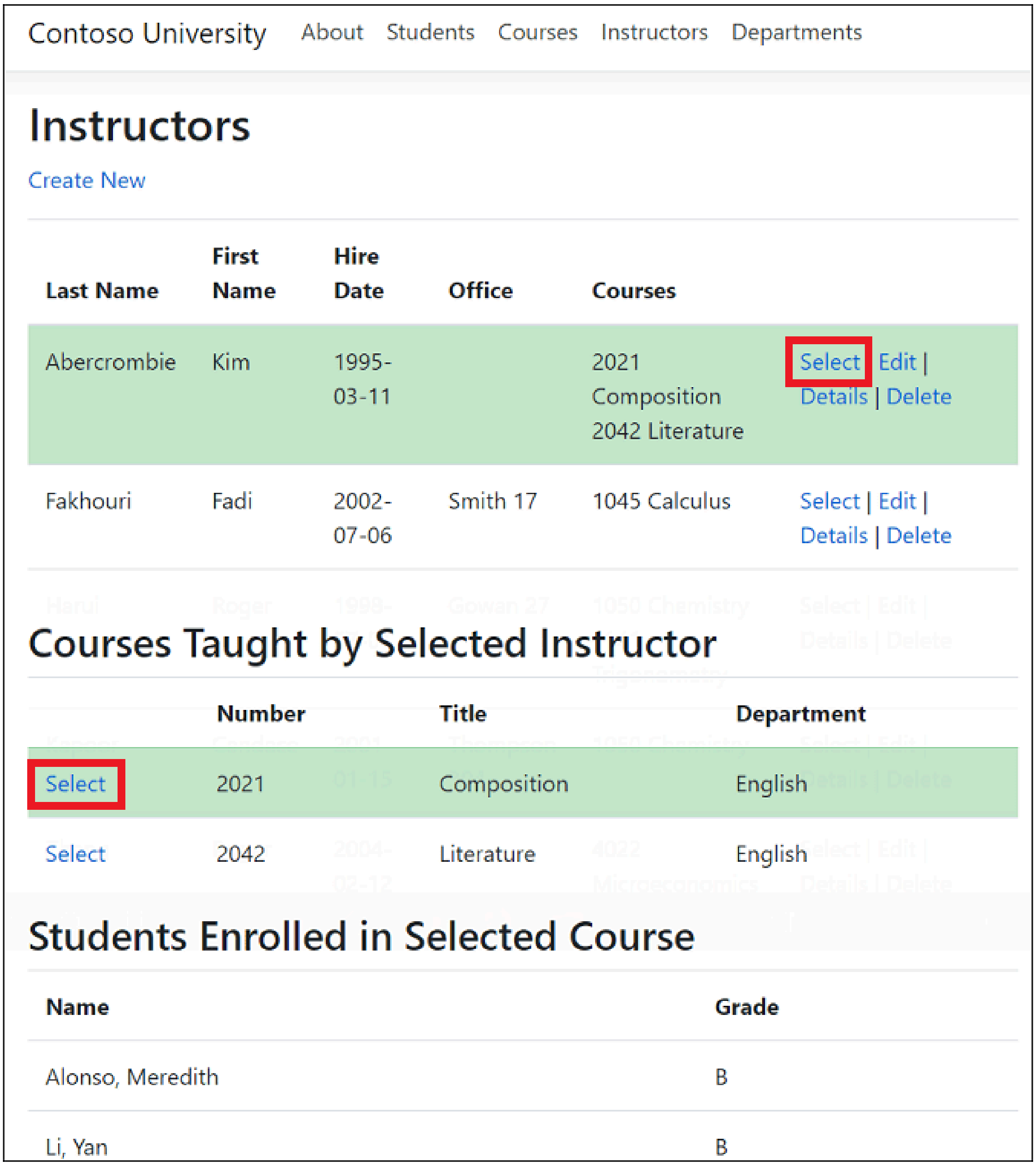
- Adds a table of courses for the selected Instructor.

- 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.



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

Next steps

The next tutorial shows how to update related data.

Part 7, Razor Pages with EF Core in ASP.NET Core - Update Related Data

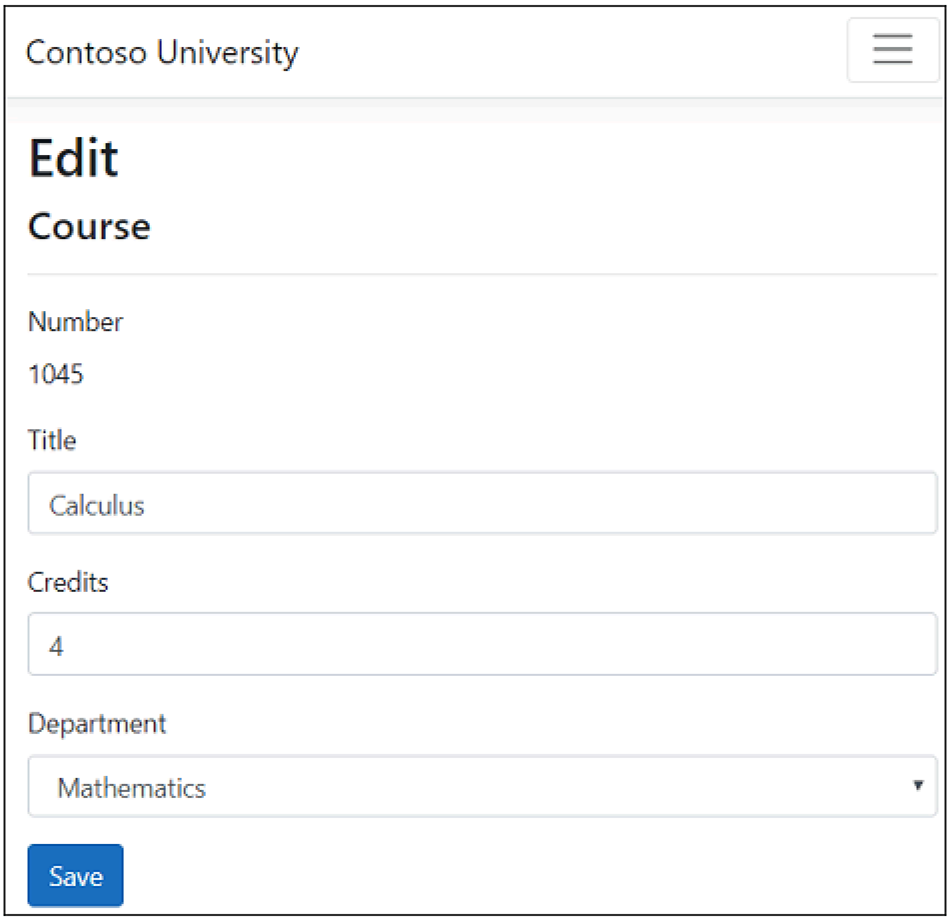
Article • 04/10/2024

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.


Contoso University

Edit

Course

Number

1045

Title

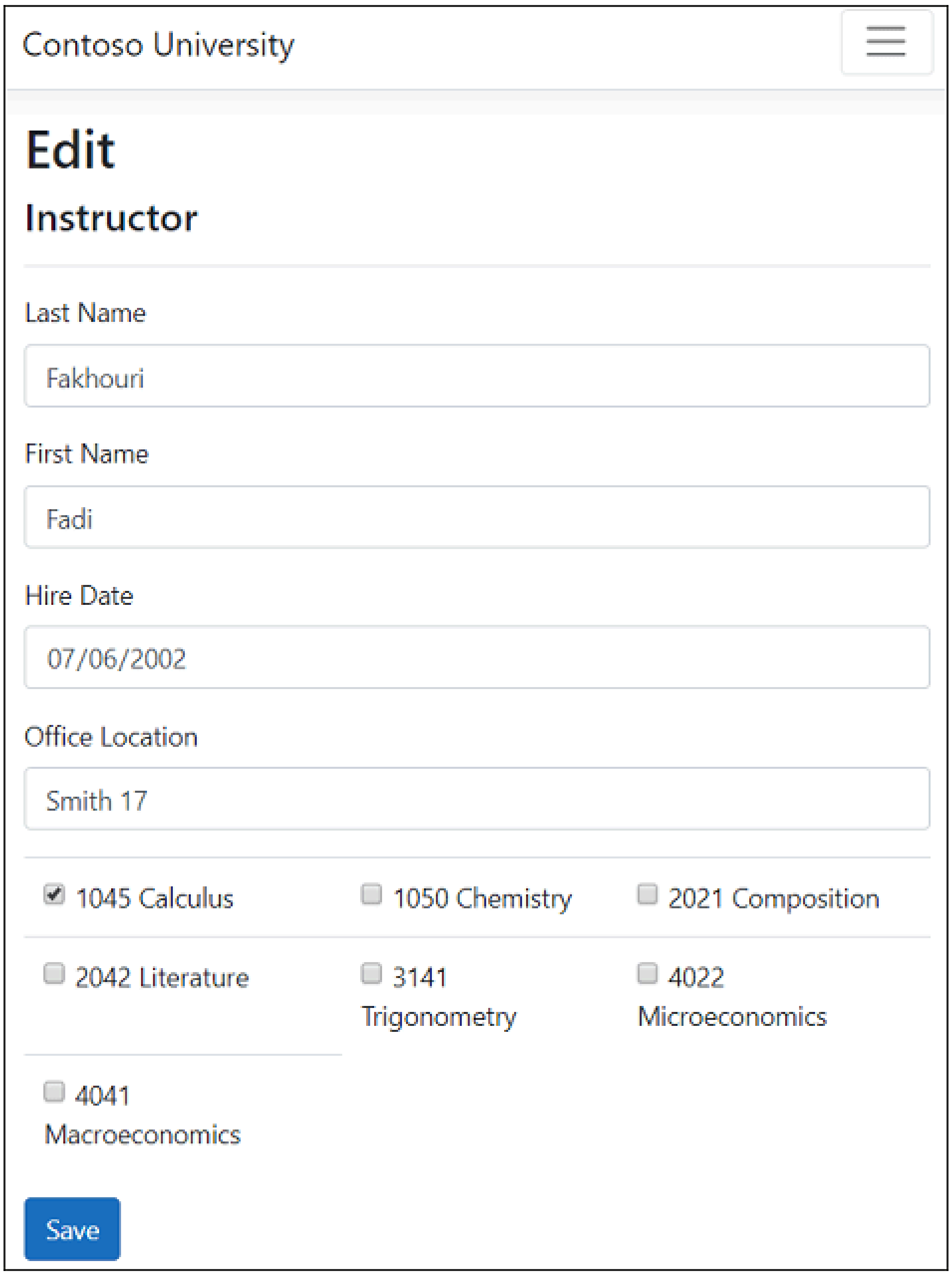
Calculus

Credits

4

Department

Mathematics


Contoso University

Edit

Instructor

Last Name

Fakhouri

First Name

Fadi

Hire Date

07/06/2002

Office Location

Smith 17

- [x] 1045 Calculus
- [ ] 1050 Chemistry
- [ ] 2021 Composition
- [ ] 2042 Literature
- [ ] 3141 Trigonometry
- [ ] 4022 Microeconomics
- [ ] 4041 Macroeconomics

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:

C#

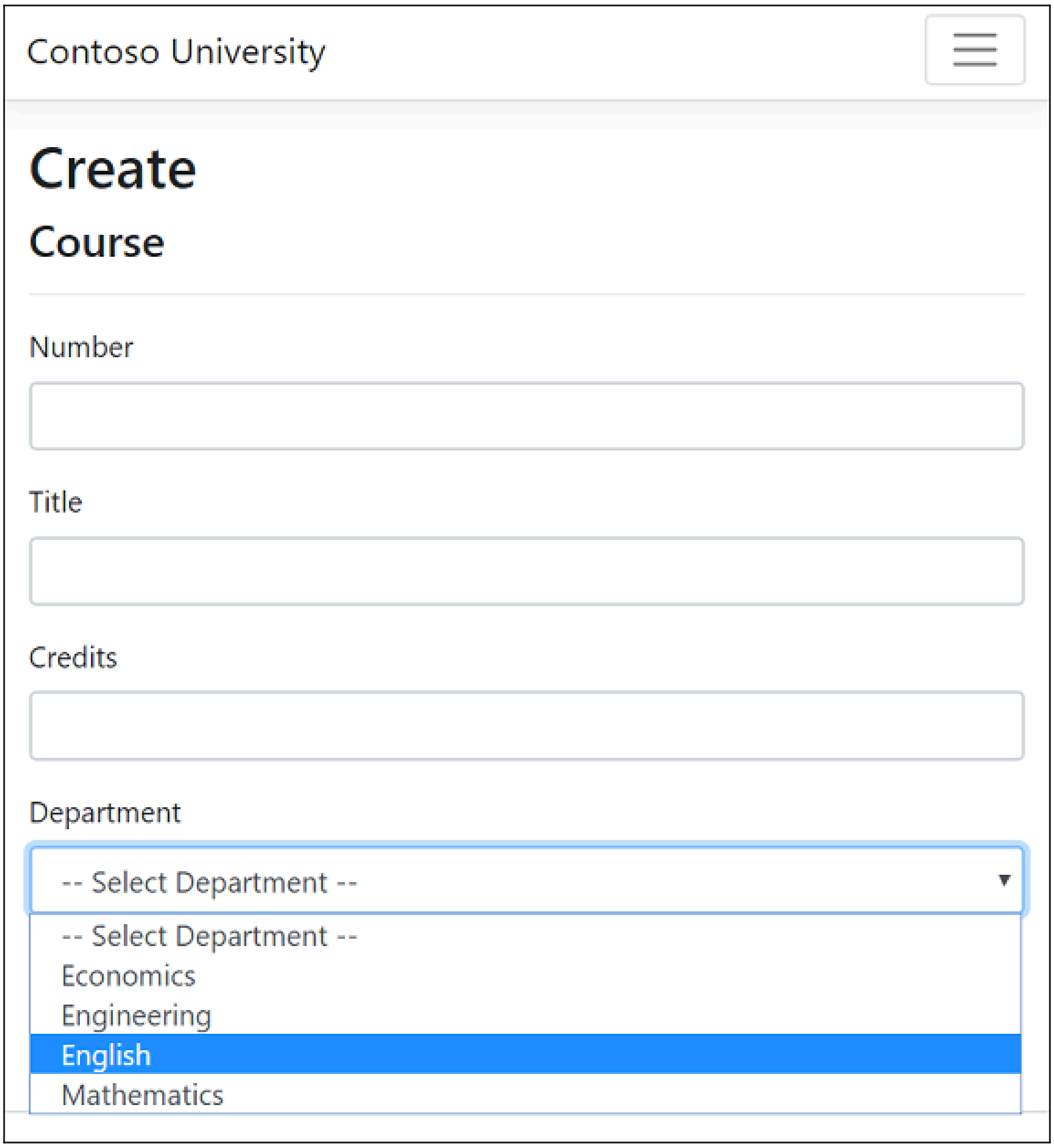
```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;

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(),
                nameof(Department.DepartmentID),
                nameof(Department.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.


Contoso University

Create

Course

Number

Title

Credits

Department

-- Select Department --

-- Select Department --
Economics
Engineering
English
Mathematics

Update `Pages/Courses/Create.cshtml.cs` with the following code:

```C#
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:

CSHTML

```
@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:

CSHTML

```
<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:

C#

```
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:

CSHTML

```
@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:

C#

```
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:

CSHTML

```
@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.

CSHTML

```
@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">
```

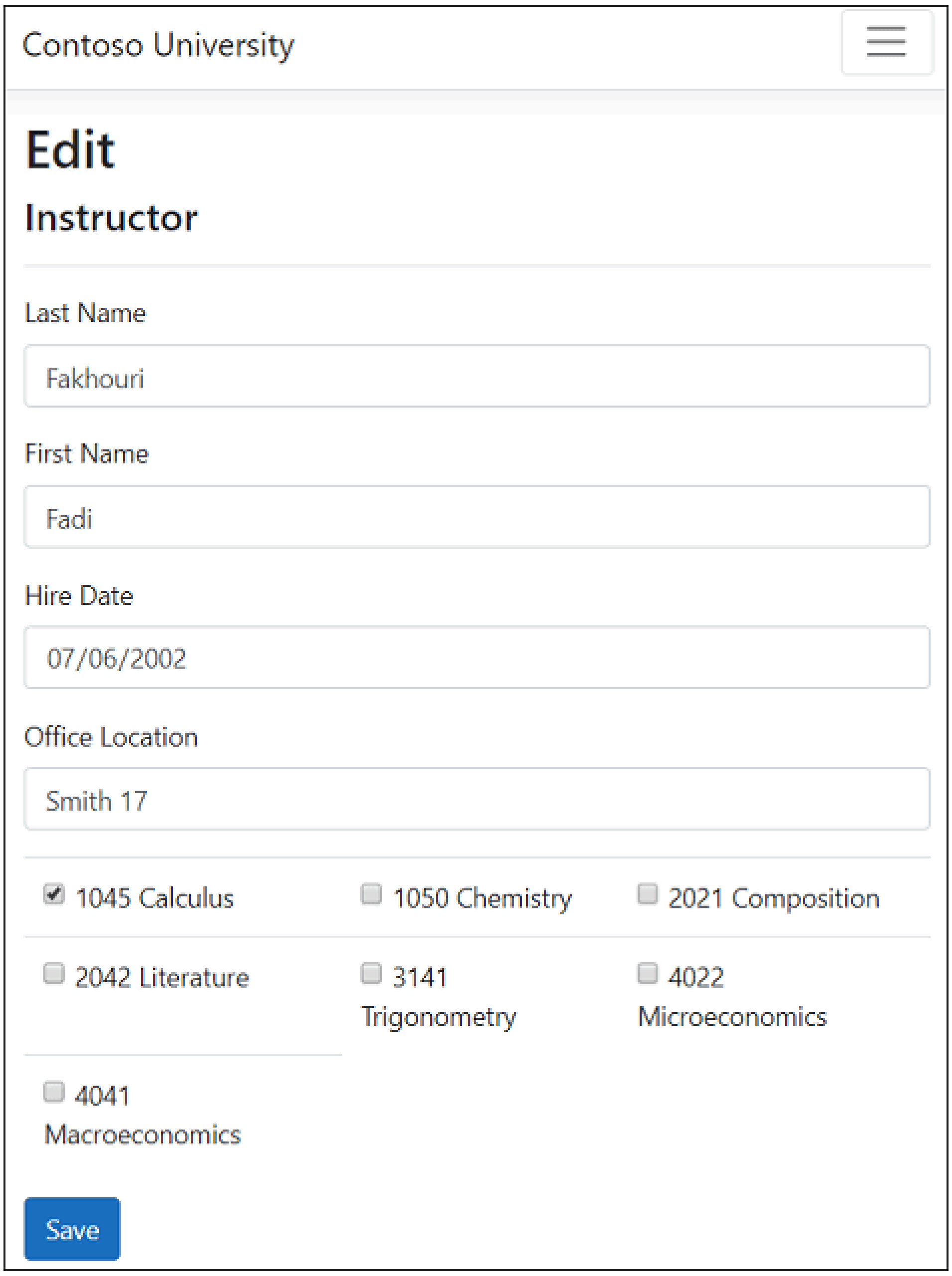
```
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Department.Name)
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Course.CourseID">Edit</a> |
    <a 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.


Contoso University

Edit

Instructor

Last Name

Fakhouri

First Name

Fadi

Hire Date

07/06/2002

Office Location

Smith 17

- [x] 1045 Calculus
- [ ] 1050 Chemistry
- [ ] 2021 Composition
- [ ] 2042 Literature
- [ ] 3141 Trigonometry
- [ ] 4022 Microeconomics
- [ ] 4041 Macroeconomics

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:

C#

```csharp
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:

```C#
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:

C#

```csharp
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,
                                            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:

```C#
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.

```C#
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.

C#

```
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:

CSHTML

```
@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:

C#

```csharp
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.

```C#
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:

CSHTML

```
@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:

C#

```csharp
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

Part 8, Razor Pages with EF Core in ASP.NET Core - Concurrency

Article • 04/10/2024

Tom Dykstra, 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 shows how to handle conflicts when multiple users update an entity concurrently.

Concurrency conflicts

A concurrency conflict occurs when:

- A user navigates to the edit page for an entity.
- Another user updates the same entity before the first user's change is written to the database.

If concurrency detection isn't enabled, whoever updates the database last overwrites the other user's changes. If this risk is acceptable, the cost of programming for concurrency might outweigh the benefit.

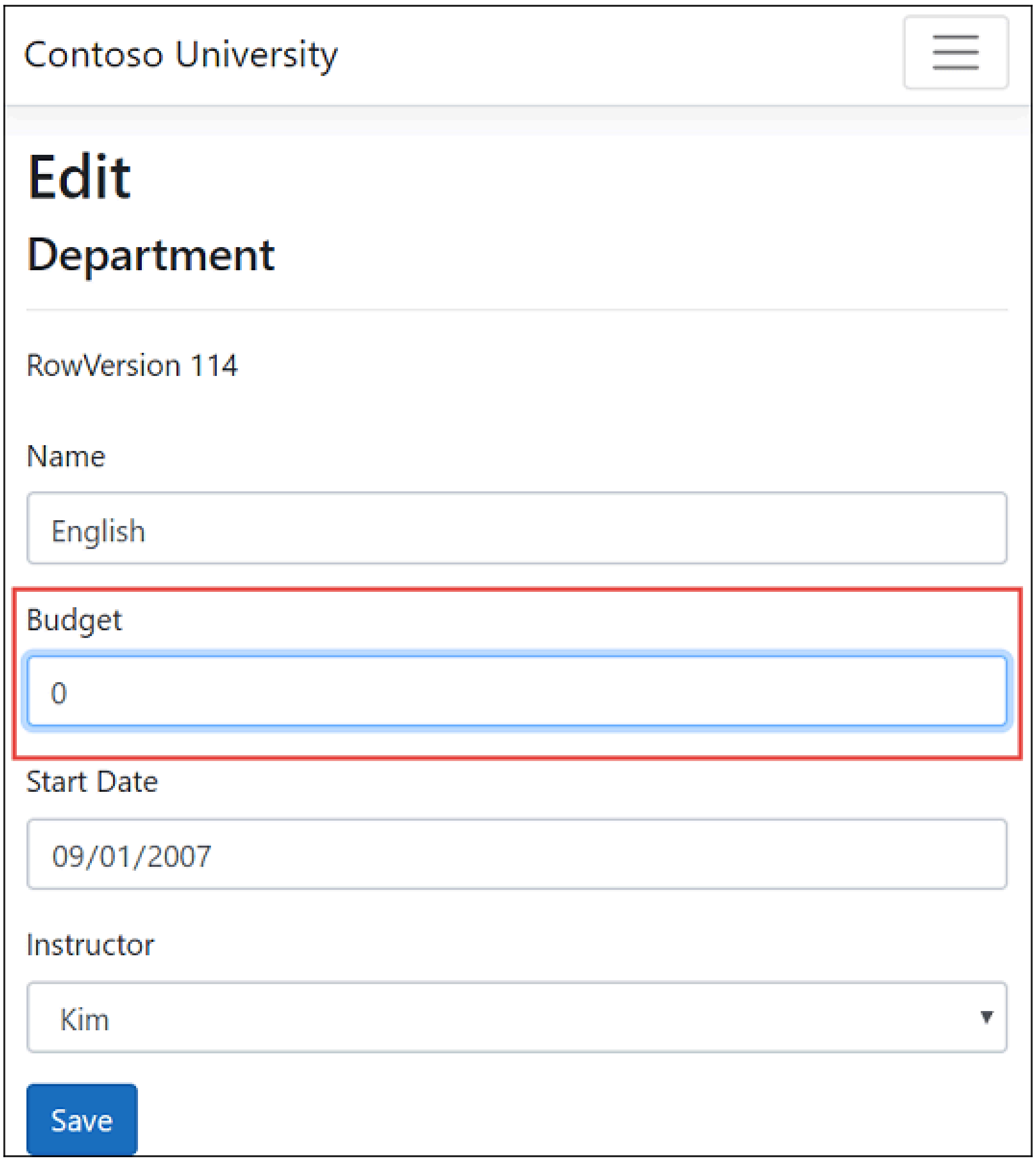
Pessimistic concurrency

One way to prevent concurrency conflicts is to use database locks. This is called pessimistic concurrency. Before the app reads a database row that it intends to update, it requests a lock. Once a row is locked for update access, no other users are allowed to lock the row until the first lock is released.

Managing locks has disadvantages. It can be complex to program and can cause performance problems as the number of users increases. Entity Framework Core provides no built-in support for pessimistic concurrency.

Optimistic concurrency

Optimistic concurrency allows concurrency conflicts to happen, and then reacts appropriately when they do. For example, Jane visits the Department edit page and changes the budget for the English department from $350,000.00 to $0.00.


Contoso University

Edit

Department

RowVersion 114

Name

English

Budget

0

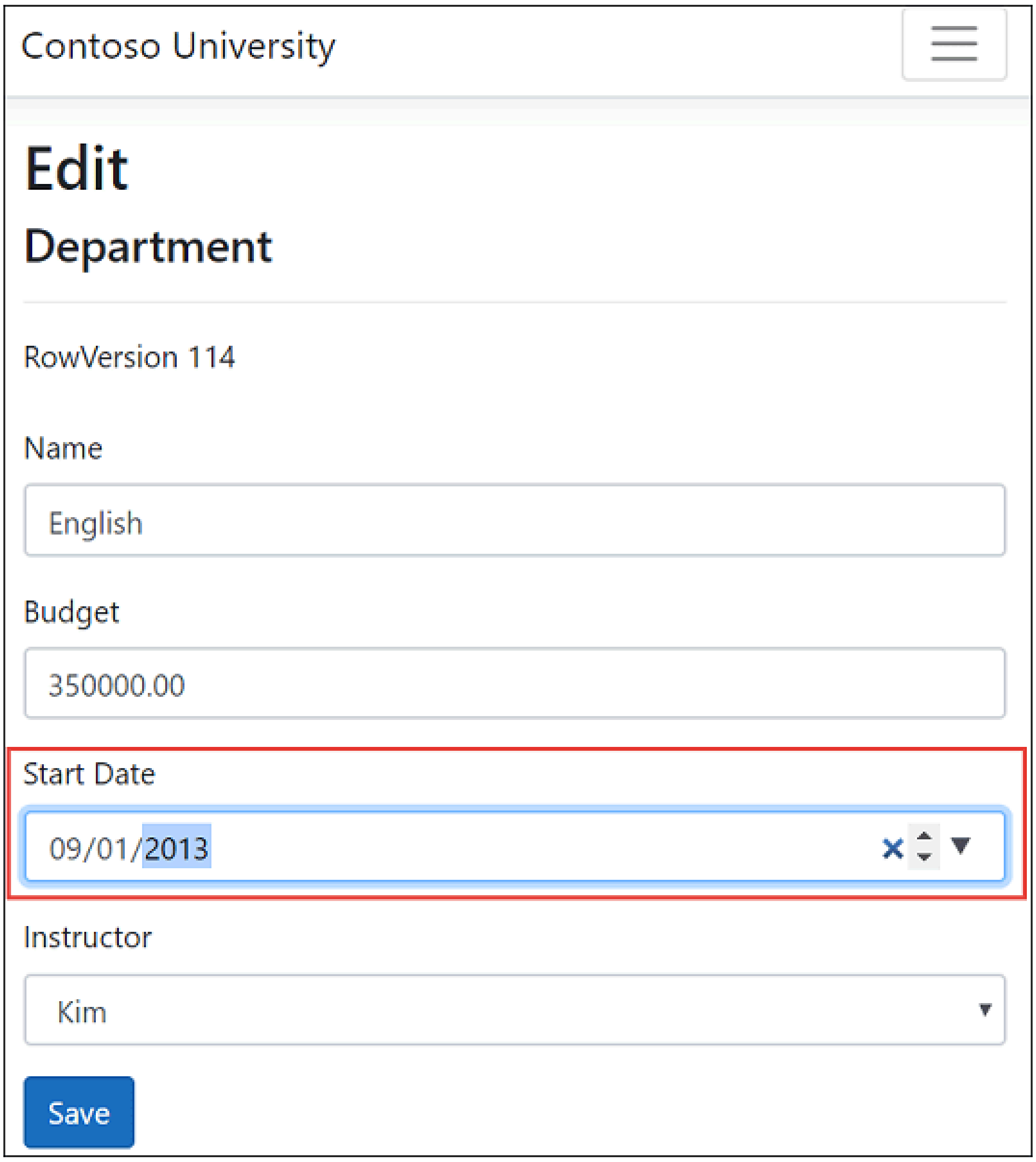
Start Date

09/01/2007

Instructor

Kim

Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.


Contoso University

Edit

Department

RowVersion 114

Name

English

Budget

350000.00

Start Date

09/01/2013

Instructor

Kim

Jane clicks **Save** first and sees her change take effect, since the browser displays the Index page with zero as the Budget amount.

John clicks **Save** on an Edit page that still shows a budget of $350,000.00. What happens next is determined by how you handle concurrency conflicts:

- Keep track of which property a user has modified and update only the corresponding columns in the database.

  In the scenario, no data would be lost. Different properties were updated by the two users. The next time someone browses the English department, they will see both Jane's and John's changes. This method of updating can reduce the number of conflicts that could result in data loss. This approach has some disadvantages:
  - Can't avoid data loss if competing changes are made to the same property.
  - Is generally not practical in a web app. It requires maintaining significant state in order to keep track of all fetched values and new values. Maintaining large amounts of state can affect app performance.

    - Can increase app complexity compared to concurrency detection on an entity.

- Let John's change overwrite Jane's change.

  The next time someone browses the English department, they will see 9/1/2013 and the fetched $350,000.00 value. This approach is called a *Client Wins* or *Last in Wins* scenario. All values from the client take precedence over what's in the data store. The scaffolded code does no concurrency handling, Client Wins happens automatically.

- Prevent John's change from being updated in the database. Typically, the app would:
    - Display an error message.
    - Show the current state of the data.
    - Allow the user to reapply the changes.

  This is called a *Store Wins* scenario. The data-store values take precedence over the values submitted by the client. The Store Wins scenario is used in this tutorial. This method ensures that no changes are overwritten without a user being alerted.

Conflict detection in EF Core

Properties configured as concurrency tokens are used to implement optimistic concurrency control. When an update or delete operation is triggered by SaveChanges or SaveChangesAsync, the value of the concurrency token in the database is compared against the original value read by EF Core:

- If the values match, the operation can complete.
- If the values do not match, EF Core assumes that another user has performed a conflicting operation, aborts the current transaction, and throws a DbUpdateConcurrencyException.

Another user or process performing an operation that conflicts with the current operation is known as *concurrency conflict*.

On relational databases EF Core checks for the value of the concurrency token in the `WHERE` clause of `UPDATE` and `DELETE` statements to detect a concurrency conflict.

The data model must be configured to enable conflict detection by including a tracking column that can be used to determine when a row has been changed. EF provides two approaches for concurrency tokens:

- Applying [ConcurrencyCheck] or IsConcurrencyToken to a property on the model. This approach is not recommended. For more information, see Concurrency Tokens in EF Core.
- Applying TimestampAttribute or IsRowVersion to a concurrency token in the model. This is the approach used in this tutorial.

The SQL Server approach and SQLite implementation details are slightly different. A difference file is shown later in the tutorial listing the differences. The Visual Studio tab shows the SQL Server approach. The Visual Studio Code tab shows the approach for non-SQL Server databases, such as SQLite.

Visual Studio

- In the model, include a tracking column that is used to determine when a row has been changed.
- Apply the TimestampAttribute to the concurrency property.

Update the `Models/Department.cs` file with the following highlighted code:

C#

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

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

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

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

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

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] ConcurrencyToken { get; set; }

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

The TimestampAttribute is what identifies the column as a concurrency tracking column. The fluent API is an alternative way to specify the tracking property:

C#

```csharp
modelBuilder.Entity<Department>()
  .Property<byte[]>("ConcurrencyToken")
  .IsRowVersion();
```

The [Timestamp] attribute on an entity property generates the following code in the ModelBuilder method:

C#

```csharp
 b.Property<byte[]>("ConcurrencyToken")
     .IsConcurrencyToken()
     .ValueGeneratedOnAddOrUpdate()
     .HasColumnType("rowversion");
```

The preceding code:

- Sets the property type `ConcurrencyToken` to byte array. `byte[]` is the required type for SQL Server.
- Calls IsConcurrencyToken. `IsConcurrencyToken` configures the property as a concurrency token. On updates, the concurrency token value in the database is compared to the original value to ensure it has not changed since the instance was retrieved from the database. If it has changed, a DbUpdateConcurrencyException is thrown and changes are not applied.
- Calls ValueGeneratedOnAddOrUpdate, which configures the `ConcurrencyToken` property to have a value automatically generated when adding or updating an entity.
- `HasColumnType("rowversion")` sets the column type in the SQL Server database to rowversion.

The following code shows a portion of the T-SQL generated by EF Core when the `Department` name is updated:

SQL

```sql
SET NOCOUNT ON;
UPDATE [Departments] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [ConcurrencyToken] = @p2;
SELECT [ConcurrencyToken]
FROM [Departments]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

The preceding highlighted code shows the `WHERE` clause containing `ConcurrencyToken`. If the database `ConcurrencyToken` doesn't equal the `ConcurrencyToken` parameter `@p2`, no rows are updated.

The following highlighted code shows the T-SQL that verifies exactly one row was updated:

SQL

```sql
SET NOCOUNT ON;
UPDATE [Departments] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [ConcurrencyToken] = @p2;
SELECT [ConcurrencyToken]
FROM [Departments]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

@@ROWCOUNT returns the number of rows affected by the last statement. If no rows are updated, EF Core throws a `DbUpdateConcurrencyException`.

Add a migration

Adding the `ConcurrencyToken` property changes the data model, which requires a migration.

Build the project.

Visual Studio

Run the following commands in the PMC:

PowerShell

```powershell
Add-Migration RowVersion
Update-Database
```

The preceding commands:

- Creates the `Migrations/{time stamp}_RowVersion.cs` migration file.
- Updates the `Migrations/SchoolContextModelSnapshot.cs` file. The update adds the following code to the `BuildModel` method:

C#

```
b.Property<byte[]>("ConcurrencyToken")
    .IsConcurrencyToken()
    .ValueGeneratedOnAddOrUpdate()
    .HasColumnType("rowversion");
```

Scaffold Department pages

Visual Studio

Follow the instructions in Scaffold Student pages with the following exceptions:

- Create a *Pages/Departments* folder.
- Use `Department` for the model class.
- Use the existing context class instead of creating a new one.

Add a utility class

In the project folder, create the `Utility` class with the following code:

Visual Studio

C#

```
namespace ContosoUniversity
{
    public static class Utility
    {
        public static string GetLastChars(byte[] token)
        {
            return token[7].ToString();
        }
    }
}
```

The `Utility` class provides the `GetLastChars` method used to display the last few characters of the concurrency token. The following code shows the code that works with both SQLite ad SQL Server:

```
C#

#if SQLiteVersion
using System;

namespace ContosoUniversity
{
    public static class Utility
    {
        public static string GetLastChars(Guid token)
        {
            return token.ToString().Substring(
                                    token.ToString().Length - 3);
        }
    }
}
#else
namespace ContosoUniversity
{
    public static class Utility
    {
        public static string GetLastChars(byte[] token)
        {
            return token[7].ToString();
        }
    }
}
#endif
```

The `#if SQLiteVersion` preprocessor directive isolates the differences in the SQLite and SQL Server versions and helps:

- The author maintain one code base for both versions.
- SQLite developers deploy the app to Azure and use SQL Azure.

Build the project.

Update the Index page

The scaffolding tool created a `ConcurrencyToken` column for the Index page, but that field wouldn't be displayed in a production app. In this tutorial, the last portion of the `ConcurrencyToken` is displayed to help show how concurrency handling works. The last portion isn't guaranteed to be unique by itself.

Update *Pages\Departments\Index.cshtml* page:

- Replace Index with Departments.
- Change the code containing `ConcurrencyToken` to show just the last few characters.
- Replace `FirstMidName` with `FullName`.

The following code shows the updated page:

CSHTML

```cshtml
@page
@model ContosoUniversity.Pages.Departments.IndexModel

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

<h2>Departments</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Department[0].Administrator)
            </th>
            <th>
                Token
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Department)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem =>
item.Administrator.FullName)
                </td>
                <td>
                    @Utility.GetLastChars(item.ConcurrencyToken)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-
id="@item.DepartmentID">Edit</a> |
                    <a asp-page="./Details" asp-route-
id="@item.DepartmentID">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Update the Edit page model

Update `Pages/Departments/Edit.cshtml.cs` with the following code:

Visual Studio

C#

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

namespace ContosoUniversity.Pages.Departments
{
    public class EditModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

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

        [BindProperty]
        public Department Department { get; set; }
        // Replace ViewData["InstructorID"]
        public SelectList InstructorNameSL { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)  // eager loading
                .AsNoTracking()                 // tracking not required
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

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

            // Use strongly typed data rather than ViewData.
            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FirstMidName");

            return Page();
        }

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

            // Fetch current department from DB.
            // ConcurrencyToken may have changed.
            var departmentToUpdate = await _context.Departments
                .Include(i => i.Administrator)
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (departmentToUpdate == null)
            {
                return HandleDeletedDepartment();
            }

            // Set ConcurrencyToken to value read in OnGetAsync
            _context.Entry(departmentToUpdate).Property(
                 d => d.ConcurrencyToken).OriginalValue =
Department.ConcurrencyToken;

            if (await TryUpdateModelAsync<Department>(
                departmentToUpdate,
                "Department",
                s => s.Name, s => s.StartDate, s => s.Budget, s =>
s.InstructorID))
            {
                try
                {
                    await _context.SaveChangesAsync();
                    return RedirectToPage("./Index");
                }
                catch (DbUpdateConcurrencyException ex)
                {
                    var exceptionEntry = ex.Entries.Single();
                    var clientValues =
(Department)exceptionEntry.Entity;
                    var databaseEntry =
exceptionEntry.GetDatabaseValues();
                    if (databaseEntry == null)
                    {
                        ModelState.AddModelError(string.Empty, "Unable
to save. " +
                            "The department was deleted by another
user.");
                        return Page();
                    }

                    var dbValues = (Department)databaseEntry.ToObject();
                    await SetDbErrorMessage(dbValues, clientValues,
_context);

                    // Save the current ConcurrencyToken so next
postback
                    // matches unless an new concurrency issue happens.
                    Department.ConcurrencyToken =
(byte[])dbValues.ConcurrencyToken;
                    // Clear the model error for the next postback.
                    ModelState.Remove($"{nameof(Department)}.
{nameof(Department.ConcurrencyToken)}");
                }
            }

            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FullName", departmentToUpdate.InstructorID);

            return Page();
        }

        private IActionResult HandleDeletedDepartment()
        {
            // ModelState contains the posted data because of the
deletion error
            // and overides the Department instance values when
displaying Page().
            ModelState.AddModelError(string.Empty,
                "Unable to save. The department was deleted by another
user.");
            InstructorNameSL = new SelectList(_context.Instructors,
"ID", "FullName", Department.InstructorID);
            return Page();
        }
        private async Task SetDbErrorMessage(Department dbValues,
                Department clientValues, SchoolContext context)
        {

            if (dbValues.Name != clientValues.Name)
            {
                ModelState.AddModelError("Department.Name",
                    $"Current value: {dbValues.Name}");
            }
            if (dbValues.Budget != clientValues.Budget)
            {
                ModelState.AddModelError("Department.Budget",
                    $"Current value: {dbValues.Budget:c}");
            }
            if (dbValues.StartDate != clientValues.StartDate)
            {
                ModelState.AddModelError("Department.StartDate",
                    $"Current value: {dbValues.StartDate:d}");
            }
            if (dbValues.InstructorID != clientValues.InstructorID)
            {
                Instructor dbInstructor = await _context.Instructors
                   .FindAsync(dbValues.InstructorID);
                ModelState.AddModelError("Department.InstructorID",
                    $"Current value: {dbInstructor?.FullName}");
            }

            ModelState.AddModelError(string.Empty,
                "The record you attempted to edit "
              + "was modified by another user after you. The "
              + "edit operation was canceled and the current values in
the database "
              + "have been displayed. If you still want to edit this
record, click "
              + "the Save button again.");
        }
    }
}
```

The concurrency updates

OriginalValue is updated with the `ConcurrencyToken` value from the entity when it was fetched in the `OnGetAsync` method. EF Core generates a `SQL UPDATE` command with a `WHERE` clause containing the original `ConcurrencyToken` value. If no rows are affected by the `UPDATE` command, a `DbUpdateConcurrencyException` exception is thrown. No rows are affected by the `UPDATE` command when no rows have the original `ConcurrencyToken` value.

Visual Studio

C#

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

    // Fetch current department from DB.
    // ConcurrencyToken may have changed.
    var departmentToUpdate = await _context.Departments
        .Include(i => i.Administrator)
        .FirstOrDefaultAsync(m => m.DepartmentID == id);

    if (departmentToUpdate == null)
    {
        return HandleDeletedDepartment();
    }

    // Set ConcurrencyToken to value read in OnGetAsync
    _context.Entry(departmentToUpdate).Property(
         d => d.ConcurrencyToken).OriginalValue =
Department.ConcurrencyToken;
```

In the preceding highlighted code:

- The value in `Department.ConcurrencyToken` is the value when the entity was fetched in the `Get` request for the `Edit` page. The value is provided to the `OnPost` method by a hidden field in the Razor page that displays the entity to be edited. The hidden field value is copied to `Department.ConcurrencyToken` by the model binder.
- `OriginalValue` is what EF Core uses in the `WHERE` clause. Before the highlighted line of code executes:
  - `OriginalValue` has the value that was in the database when `FirstOrDefaultAsync` was called in this method.
  - This value might be different from what was displayed on the Edit page.
- The highlighted code makes sure that EF Core uses the original `ConcurrencyToken` value from the displayed `Department` entity in the SQL `UPDATE` statement's `WHERE` clause.

The following code shows the `Department` model. `Department` is initialized in the:

- `OnGetAsync` method by the EF query.

- `OnPostAsync` method by the hidden field in the Razor page using model binding:

Visual Studio

C#

```
public class EditModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

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

    [BindProperty]
    public Department Department { get; set; }
    // Replace ViewData["InstructorID"]
    public SelectList InstructorNameSL { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Department = await _context.Departments
            .Include(d => d.Administrator)  // eager loading
            .AsNoTracking()                 // tracking not required
            .FirstOrDefaultAsync(m => m.DepartmentID == id);

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

        // Use strongly typed data rather than ViewData.
        InstructorNameSL = new SelectList(_context.Instructors,
            "ID", "FirstMidName");

        return Page();
    }

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

        // Fetch current department from DB.
        // ConcurrencyToken may have changed.
        var departmentToUpdate = await _context.Departments
            .Include(i => i.Administrator)
            .FirstOrDefaultAsync(m => m.DepartmentID == id);

        if (departmentToUpdate == null)
```

```
          {
              return HandleDeletedDepartment();
          }

          // Set ConcurrencyToken to value read in OnGetAsync
          _context.Entry(departmentToUpdate).Property(
               d => d.ConcurrencyToken).OriginalValue =
  Department.ConcurrencyToken;
```

The preceding code shows the `ConcurrencyToken` value of the `Department` entity from the `HTTP POST` request is set to the `ConcurrencyToken` value from the `HTTP GET` request.

When a concurrency error happens, the following highlighted code gets the client values (the values posted to this method) and the database values.

C#

```
if (await TryUpdateModelAsync<Department>(
    departmentToUpdate,
    "Department",
    s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
{
    try
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var exceptionEntry = ex.Entries.Single();
        var clientValues = (Department)exceptionEntry.Entity;
        var databaseEntry = exceptionEntry.GetDatabaseValues();
        if (databaseEntry == null)
        {
            ModelState.AddModelError(string.Empty, "Unable to save. " +
                "The department was deleted by another user.");
            return Page();
        }

        var dbValues = (Department)databaseEntry.ToObject();
        await SetDbErrorMessage(dbValues, clientValues, _context);

        // Save the current ConcurrencyToken so next postback
        // matches unless an new concurrency issue happens.
        Department.ConcurrencyToken = dbValues.ConcurrencyToken;
        // Clear the model error for the next postback.
        ModelState.Remove($"{nameof(Department)}.
{nameof(Department.ConcurrencyToken)}");
    }
```

The following code adds a custom error message for each column that has database values different from what was posted to `OnPostAsync`:

C#

```
private async Task SetDbErrorMessage(Department dbValues,
        Department clientValues, SchoolContext context)
{

    if (dbValues.Name != clientValues.Name)
    {
        ModelState.AddModelError("Department.Name",
            $"Current value: {dbValues.Name}");
    }
    if (dbValues.Budget != clientValues.Budget)
    {
        ModelState.AddModelError("Department.Budget",
            $"Current value: {dbValues.Budget:c}");
    }
    if (dbValues.StartDate != clientValues.StartDate)
    {
        ModelState.AddModelError("Department.StartDate",
            $"Current value: {dbValues.StartDate:d}");
    }
    if (dbValues.InstructorID != clientValues.InstructorID)
    {
        Instructor dbInstructor = await _context.Instructors
           .FindAsync(dbValues.InstructorID);
        ModelState.AddModelError("Department.InstructorID",
            $"Current value: {dbInstructor?.FullName}");
    }

    ModelState.AddModelError(string.Empty,
        "The record you attempted to edit "
      + "was modified by another user after you. The "
      + "edit operation was canceled and the current values in the database
"
      + "have been displayed. If you still want to edit this record, click "
      + "the Save button again.");
}
```

The following highlighted code sets the `ConcurrencyToken` value to the new value retrieved from the database. The next time the user clicks **Save**, only concurrency errors that happen since the last display of the Edit page will be caught.

C#

```
if (await TryUpdateModelAsync<Department>(
    departmentToUpdate,
    "Department",
    s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
```

```
{
    try
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var exceptionEntry = ex.Entries.Single();
        var clientValues = (Department)exceptionEntry.Entity;
        var databaseEntry = exceptionEntry.GetDatabaseValues();
        if (databaseEntry == null)
        {
            ModelState.AddModelError(string.Empty, "Unable to save. " +
                "The department was deleted by another user.");
            return Page();
        }

        var dbValues = (Department)databaseEntry.ToObject();
        await SetDbErrorMessage(dbValues, clientValues, _context);

        // Save the current ConcurrencyToken so next postback
        // matches unless an new concurrency issue happens.
        Department.ConcurrencyToken = dbValues.ConcurrencyToken;
        // Clear the model error for the next postback.
        ModelState.Remove($"{nameof(Department)}.
{nameof(Department.ConcurrencyToken)}");
    }
```

The ModelState.Remove statement is required because `ModelState` has the previous `ConcurrencyToken` value. In the Razor Page, the `ModelState` value for a field takes precedence over the model property values when both are present.

SQL Server vs SQLite code differences

The following shows the differences between the SQL Server and SQLite versions:

```diff
+ using System;    // For GUID on SQLite

+ departmentToUpdate.ConcurrencyToken = Guid.NewGuid();

 _context.Entry(departmentToUpdate)
    .Property(d => d.ConcurrencyToken).OriginalValue =
Department.ConcurrencyToken;

- Department.ConcurrencyToken = (byte[])dbValues.ConcurrencyToken;
+ Department.ConcurrencyToken = dbValues.ConcurrencyToken;
```

Update the Edit Razor page

Update `Pages/Departments/Edit.cshtml` with the following code:

CSHTML

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Department</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="Department.DepartmentID" />
            <input type="hidden" asp-for="Department.ConcurrencyToken" />
            <div class="form-group">
                <label>Version</label>
                @Utility.GetLastChars(Model.Department.ConcurrencyToken)
            </div>
            <div class="form-group">
                <label asp-for="Department.Name" class="control-label">
</label>
                <input asp-for="Department.Name" class="form-control" />
                <span asp-validation-for="Department.Name" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.Budget" class="control-label">
</label>
                <input asp-for="Department.Budget" class="form-control" />
                <span asp-validation-for="Department.Budget" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.StartDate" class="control-label">
</label>
                <input asp-for="Department.StartDate" class="form-control"
/>
                <span asp-validation-for="Department.StartDate" class="text-
danger">
                </span>
            </div>
            <div class="form-group">
                <label class="control-label">Instructor</label>
                <select asp-for="Department.InstructorID" class="form-
control"
                        asp-items="@Model.InstructorNameSL"></select>
                <span asp-validation-for="Department.InstructorID"
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:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds a hidden row version. `ConcurrencyToken` must be added so postback binds the value.
- Displays the last byte of `ConcurrencyToken` for debugging purposes.
- Replaces `ViewData` with the strongly-typed `InstructorNameSL`.

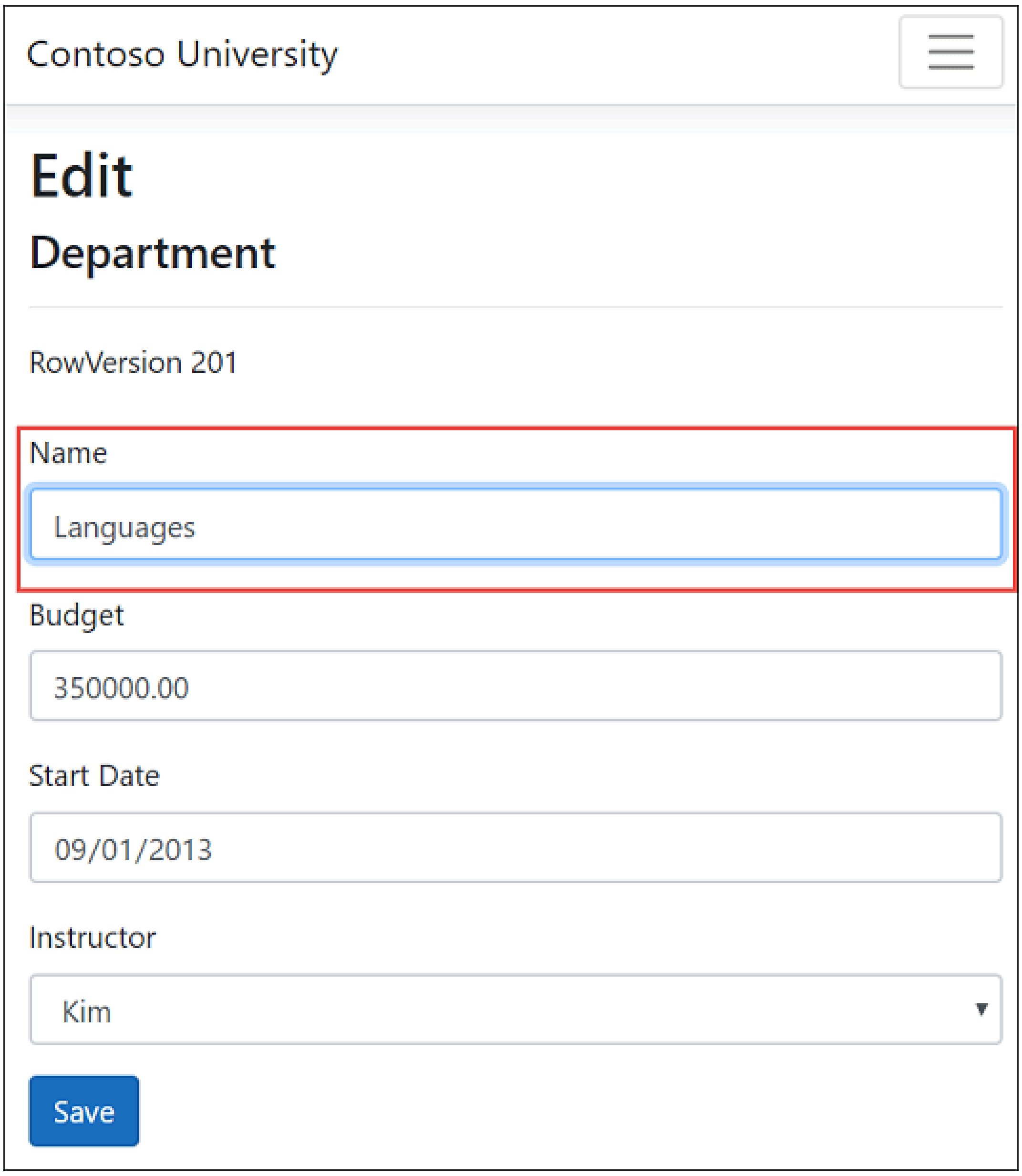
Test concurrency conflicts with the Edit page

Open two browsers instances of Edit on the English department:

- Run the app and select Departments.
- Right-click the **Edit** hyperlink for the English department and select **Open in new tab**.
- In the first tab, click the **Edit** hyperlink for the English department.

The two browser tabs display the same information.

Change the name in the first browser tab and click **Save**.


Contoso University

Edit

Department

RowVersion 201

Name: Languages

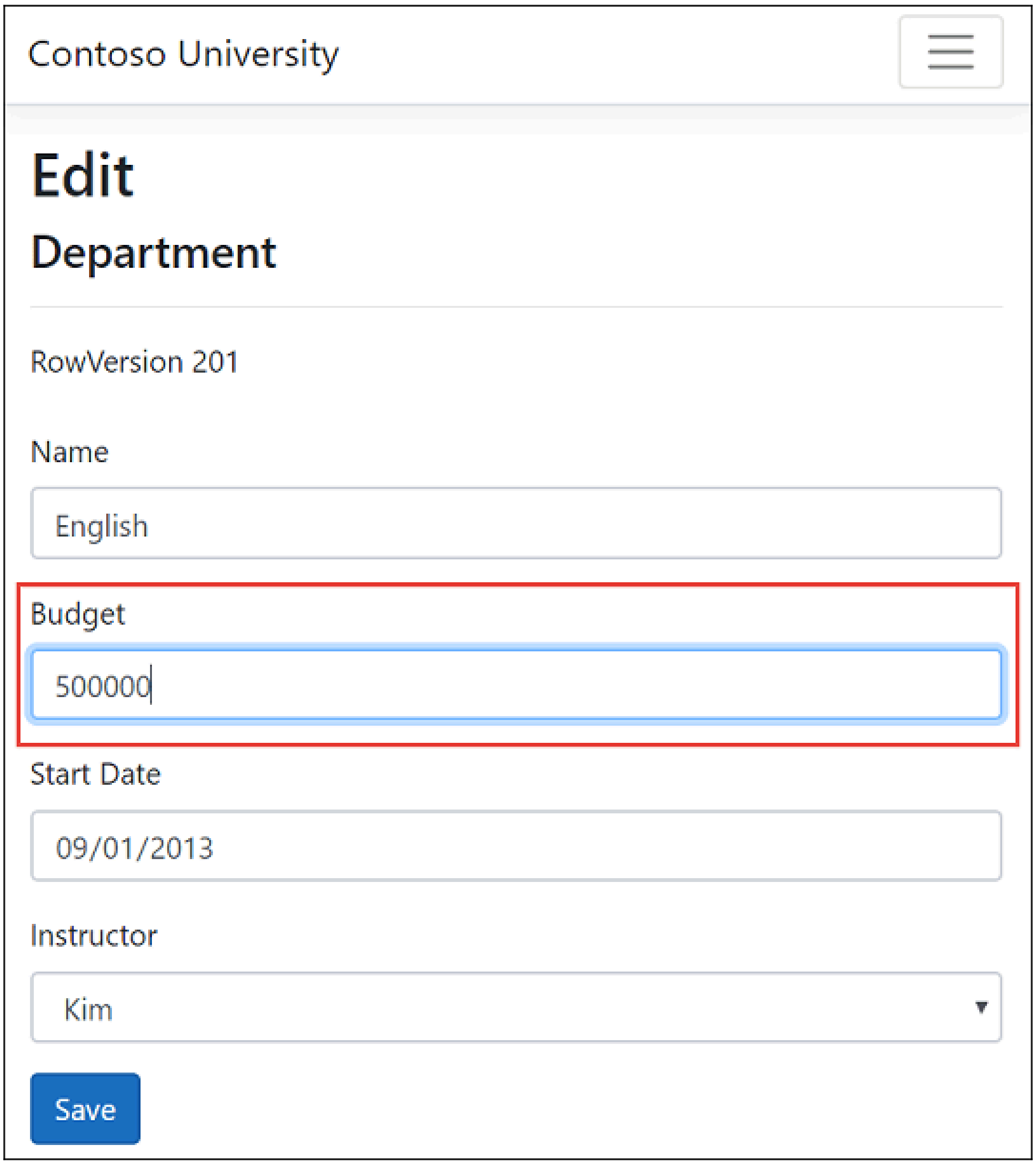
Budget: 350000.00

Start Date: 09/01/2013

Instructor: Kim

The browser shows the Index page with the changed value and updated `ConcurrencyToken` indicator. Note the updated `ConcurrencyToken` indicator, it's displayed on the second postback in the other tab.

Change a different field in the second browser tab.



Contoso University

Edit

Department

RowVersion 201

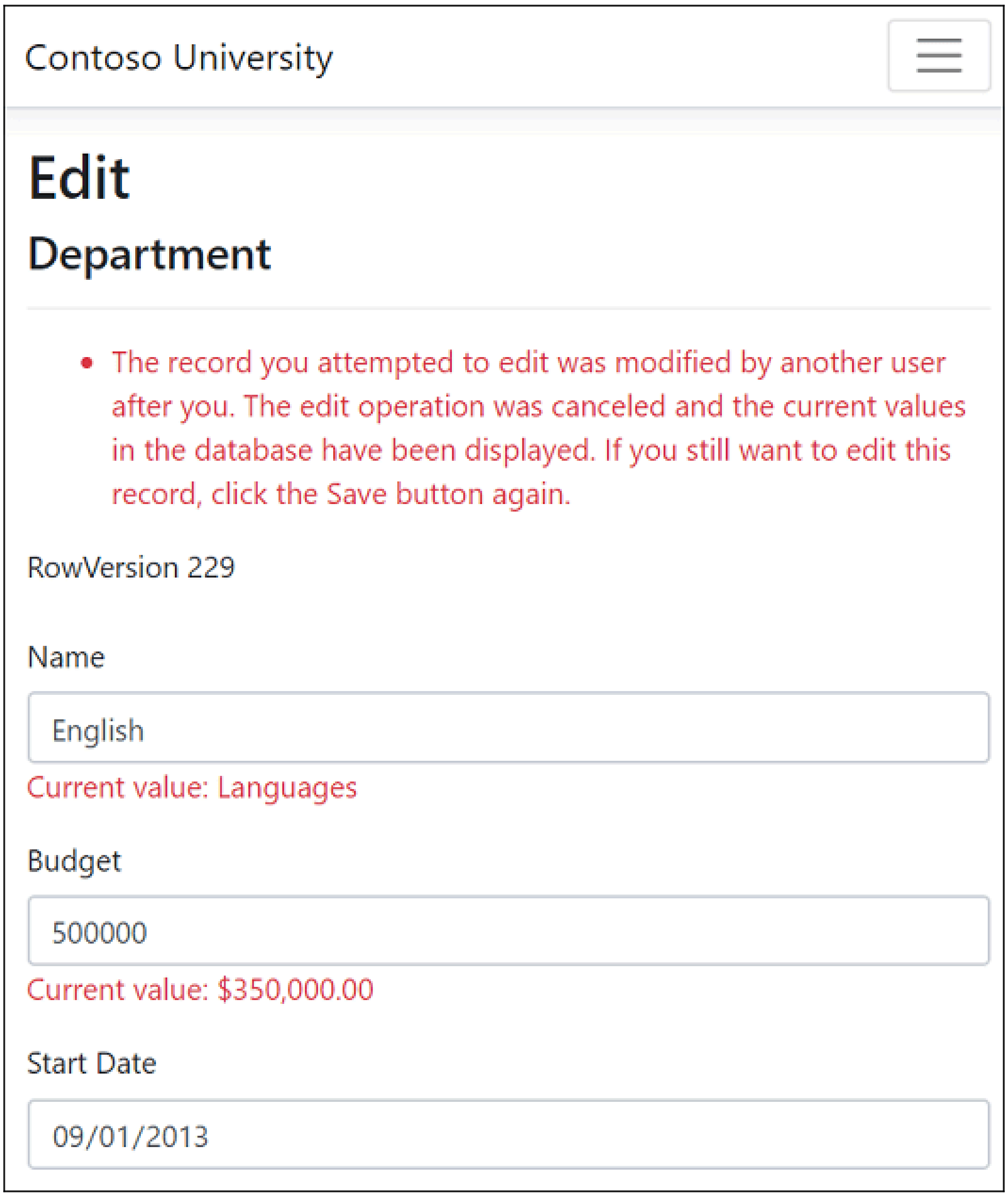
Name: English

Budget: 500000

Start Date: 09/01/2013

Instructor: Kim

Click **Save**. You see error messages for all fields that don't match the database values:


Contoso University

Edit

Department

- The record you attempted to edit was modified by another user after you. The edit operation was canceled and the current values in the database have been displayed. If you still want to edit this record, click the Save button again.

RowVersion 229

Name

English

Current value: Languages

Budget

500000

Current value: $350,000.00

Start Date

09/01/2013

This browser window didn't intend to change the Name field. Copy and paste the current value (Languages) into the Name field. Tab out. Client-side validation removes the error message.

Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values in the Index page.

Update the Delete page model

Update `Pages/Departments/Delete.cshtml.cs` with the following code:

C#

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

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

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

        [BindProperty]
        public Department Department { get; set; }
        public string ConcurrencyErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int id, bool?
concurrencyError)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

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

            if (concurrencyError.GetValueOrDefault())
            {
                ConcurrencyErrorMessage = "The record you attempted to
delete "
                  + "was modified by another user after you selected delete.
"
                  + "The delete operation was canceled and the current
values in the "
                  + "database have been displayed. If you still want to
delete this "
                  + "record, click the Delete button again.";
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            try
            {
                if (await _context.Departments.AnyAsync(
                    m => m.DepartmentID == id))
                {
                    // Department.ConcurrencyToken value is from when the
entity
```

```
                    // was fetched. If it doesn't match the DB, a
                    // DbUpdateConcurrencyException exception is thrown.
                    _context.Departments.Remove(Department);
                    await _context.SaveChangesAsync();
                }
                return RedirectToPage("./Index");
            }
            catch (DbUpdateConcurrencyException)
            {
                return RedirectToPage("./Delete",
                    new { concurrencyError = true, id = id });
            }
        }
    }
}
```

The Delete page detects concurrency conflicts when the entity has changed after it was fetched. `Department.ConcurrencyToken` is the row version when the entity was fetched. When EF Core creates the `SQL DELETE` command, it includes a WHERE clause with `ConcurrencyToken`. If the `SQL DELETE` command results in zero rows affected:

- The `ConcurrencyToken` in the `SQL DELETE` command doesn't match `ConcurrencyToken` in the database.
- A `DbUpdateConcurrencyException` exception is thrown.
- `OnGetAsync` is called with the `concurrencyError`.

Update the Delete Razor page

Update `Pages/Departments/Delete.cshtml` with the following code:

CSHTML

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.DeleteModel

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

<h1>Delete</h1>

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

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
```

```
                @Html.DisplayNameFor(model => model.Department.Name)
            </dt>
            <dd class="col-sm-10">
                @Html.DisplayFor(model => model.Department.Name)
            </dd>
            <dt class="col-sm-2">
                @Html.DisplayNameFor(model => model.Department.Budget)
            </dt>
            <dd class="col-sm-10">
                @Html.DisplayFor(model => model.Department.Budget)
            </dd>
            <dt class="col-sm-2">
                @Html.DisplayNameFor(model => model.Department.StartDate)
            </dt>
            <dd class="col-sm-10">
                @Html.DisplayFor(model => model.Department.StartDate)
            </dd>
            <dt class="col-sm-2">
                @Html.DisplayNameFor(model => model.Department.ConcurrencyToken)
            </dt>
            <dd class="col-sm-10">
                @Utility.GetLastChars(Model.Department.ConcurrencyToken)
            </dd>
            <dt class="col-sm-2">
                @Html.DisplayNameFor(model => model.Department.Administrator)
            </dt>
            <dd class="col-sm-10">
                @Html.DisplayFor(model =>
  model.Department.Administrator.FullName)
            </dd>
        </dl>

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

The preceding code makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds an error message.
- Replaces FirstMidName with FullName in the **Administrator** field.
- Changes `ConcurrencyToken` to display the last byte.
- Adds a hidden row version. `ConcurrencyToken` must be added so postback binds the value.

Test concurrency conflicts

Create a test department.

Open two browsers instances of Delete on the test department:

- Run the app and select Departments.
- Right-click the **Delete** hyperlink for the test department and select **Open in new tab**.
- Click the **Edit** hyperlink for the test department.

The two browser tabs display the same information.

Change the budget in the first browser tab and click **Save**.

The browser shows the Index page with the changed value and updated `ConcurrencyToken` indicator. Note the updated `ConcurrencyToken` indicator, it's displayed on the second postback in the other tab.

Delete the test department from the second tab. A concurrency error is display with the current values from the database. Clicking **Delete** deletes the entity, unless `ConcurrencyToken` has been updated.

Additional resources

- Concurrency Tokens in EF Core
- Handle concurrency in EF Core
- Debugging ASP.NET Core 2.x source

Next steps

This is the last tutorial in the series. Additional topics are covered in the MVC version of this tutorial series.

Previous tutorial

ASP.NET Core MVC with EF Core - tutorial series

Article • 04/10/2024

This tutorial teaches ASP.NET Core MVC and Entity Framework Core with controllers and views. Razor Pages is an alternative programming model. For new development, we recommend Razor Pages over MVC with controllers and views. See the Razor Pages version of this tutorial. Each tutorial covers some material the other doesn't:

Some things this MVC tutorial has that the Razor Pages tutorial doesn't:

- Implement inheritance in the data model
- Perform raw SQL queries
- Use dynamic LINQ to simplify code

Some things the Razor Pages tutorial has that this one doesn't:

- Use Select method to load related data
- Best practices for EF.

1. Get started
2. Create, Read, Update, and Delete operations
3. Sorting, filtering, paging, and grouping
4. Migrations
5. Create a complex data model
6. Reading related data
7. Updating related data
8. Handle concurrency conflicts
9. Inheritance
10. Advanced topics

Tutorial: Get started with EF Core in an ASP.NET MVC web app

Article • 04/10/2024

By Tom Dykstra and Rick Anderson

This tutorial teaches ASP.NET Core MVC and Entity Framework Core with controllers and views. Razor Pages is an alternative programming model. For new development, we recommend Razor Pages over MVC with controllers and views. See the Razor Pages version of this tutorial. Each tutorial covers some material the other doesn't:

Some things this MVC tutorial has that the Razor Pages tutorial doesn't:

- Implement inheritance in the data model
- Perform raw SQL queries
- Use dynamic LINQ to simplify code

Some things the Razor Pages tutorial has that this one doesn't:

- Use Select method to load related data
- Best practices for EF.

The Contoso University sample web app demonstrates how to create an ASP.NET Core MVC web app using Entity Framework (EF) Core and Visual Studio.

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

Prerequisites

- If you're new to ASP.NET Core MVC, go through the Get started with ASP.NET Core MVC tutorial series before starting this one.

- Visual Studio 2022 with the **ASP.NET and web development** workload.
- .NET 6.0 SDK

This tutorial has not been updated for ASP.NET Core 6 or later. The tutorial's instructions will not work correctly if you create a project that targets ASP.NET Core 6 or later. For example, the ASP.NET Core 6 and later web templates use the minimal hosting model, which unifies `Startup.cs` and `Program.cs` into a single `Program.cs` file.

Another difference introduced in .NET 6 is the NRT (nullable reference types) feature. The project templates enable this feature by default. Problems can happen where EF considers a property to be required in .NET 6 which is nullable in .NET 5. For example, the Create Student page will fail silently unless the `Enrollments` property is made nullable or the `asp-validation-summary` helper tag is changed from `ModelOnly` to `All`.

We recommend that you install and use the .NET 5 SDK for this tutorial. Until this tutorial is updated, see Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8 on how to use Entity Framework with ASP.NET Core 6 or later.

Database engines

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

Solve problems and troubleshoot

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

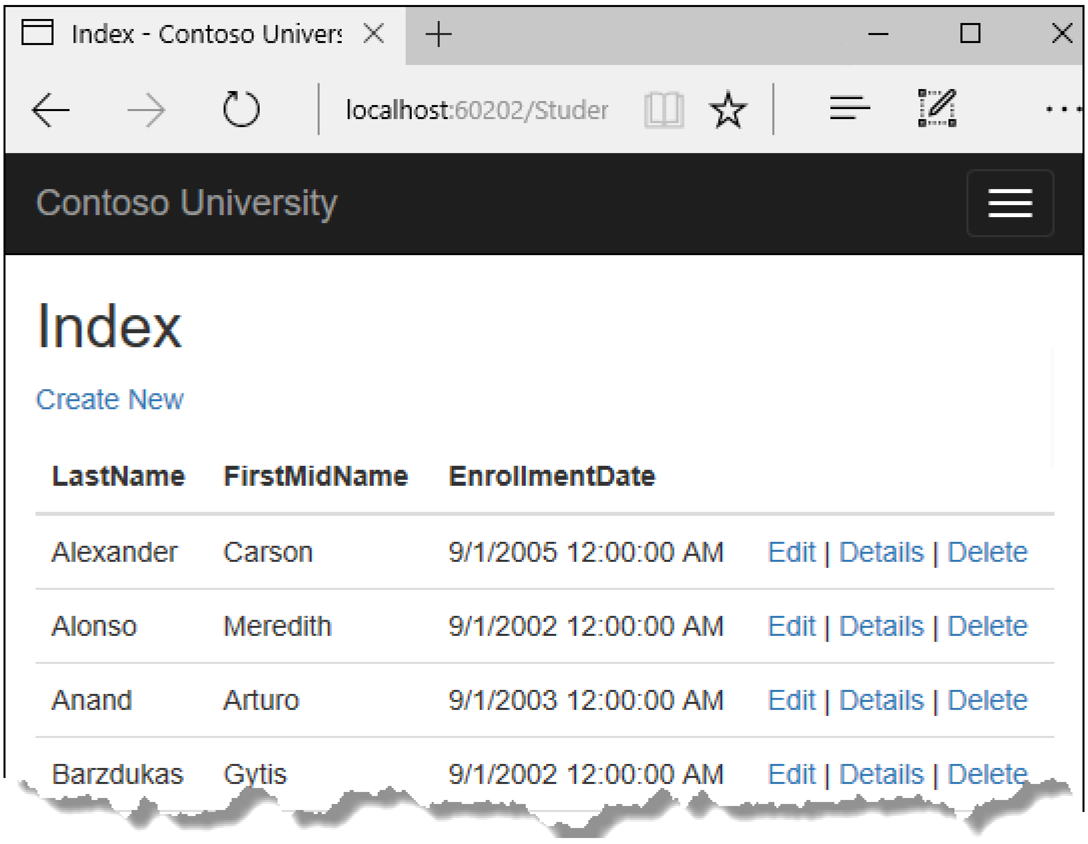
Tip

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

Contoso University web app

The app built in these tutorials is a basic university web site.

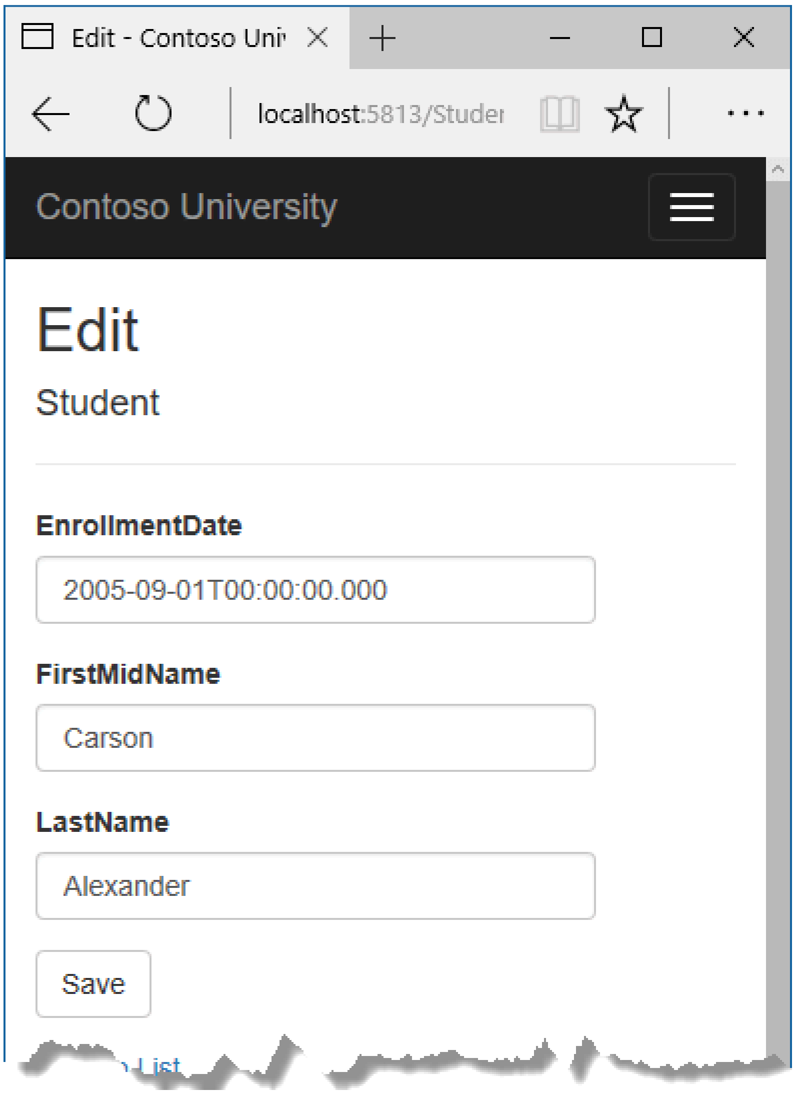
Users can view and update student, course, and instructor information. Here are a few of the screens in the app:



Index

Create New

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



Edit

Student

EnrollmentDate

2005-09-01T00:00:00.000

FirstMidName

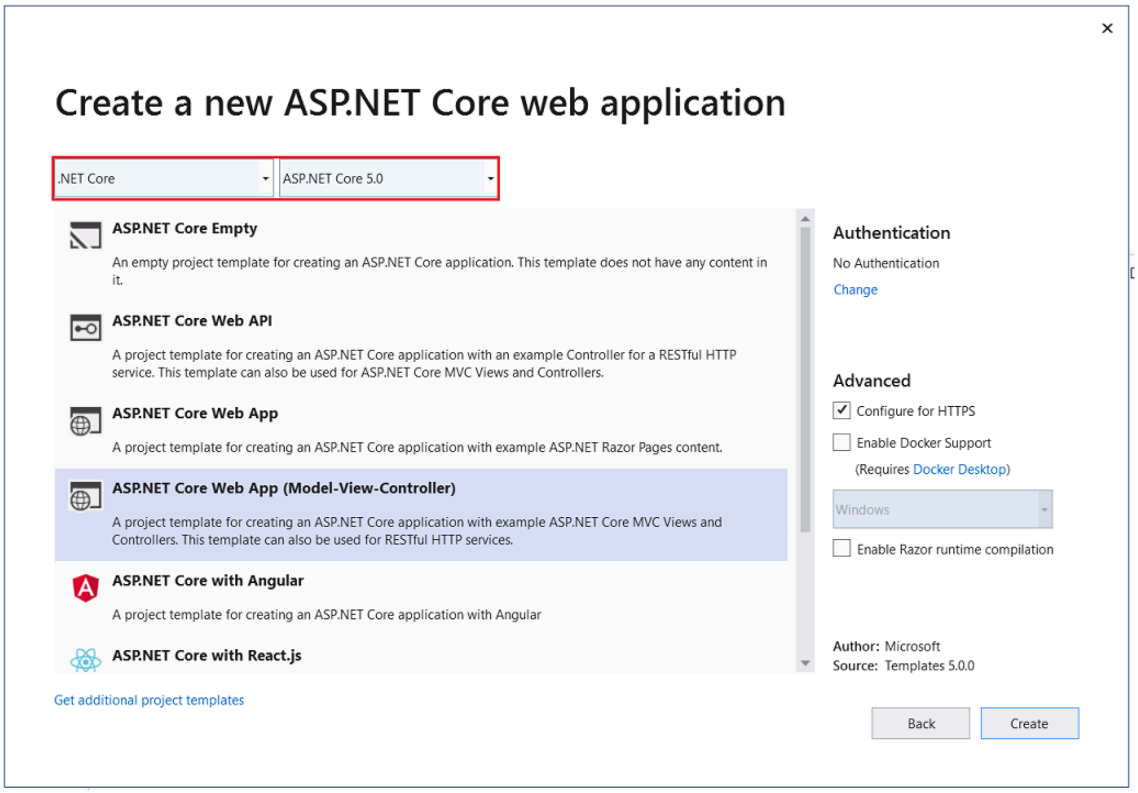
Carson

LastName

Alexander

Create web app

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 (Model-View-Controller)**.
   c. **Create**



Create a new ASP.NET Core web application

.NET Core | ASP.NET Core 5.0

ASP.NET Core Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

ASP.NET Core Web API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

ASP.NET Core Web App
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

ASP.NET Core Web App (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

ASP.NET Core with Angular
A project template for creating an ASP.NET Core application with Angular

ASP.NET Core with React.js

Authentication
No Authentication
Change

Advanced
✓ Configure for HTTPS
Enable Docker Support
(Requires Docker Desktop)
Windows
Enable Razor runtime compilation

Author: Microsoft
Source: Templates 5.0.0

Set up the site style

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

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

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

The preceding changes are highlighted in the following code:

CSHTML

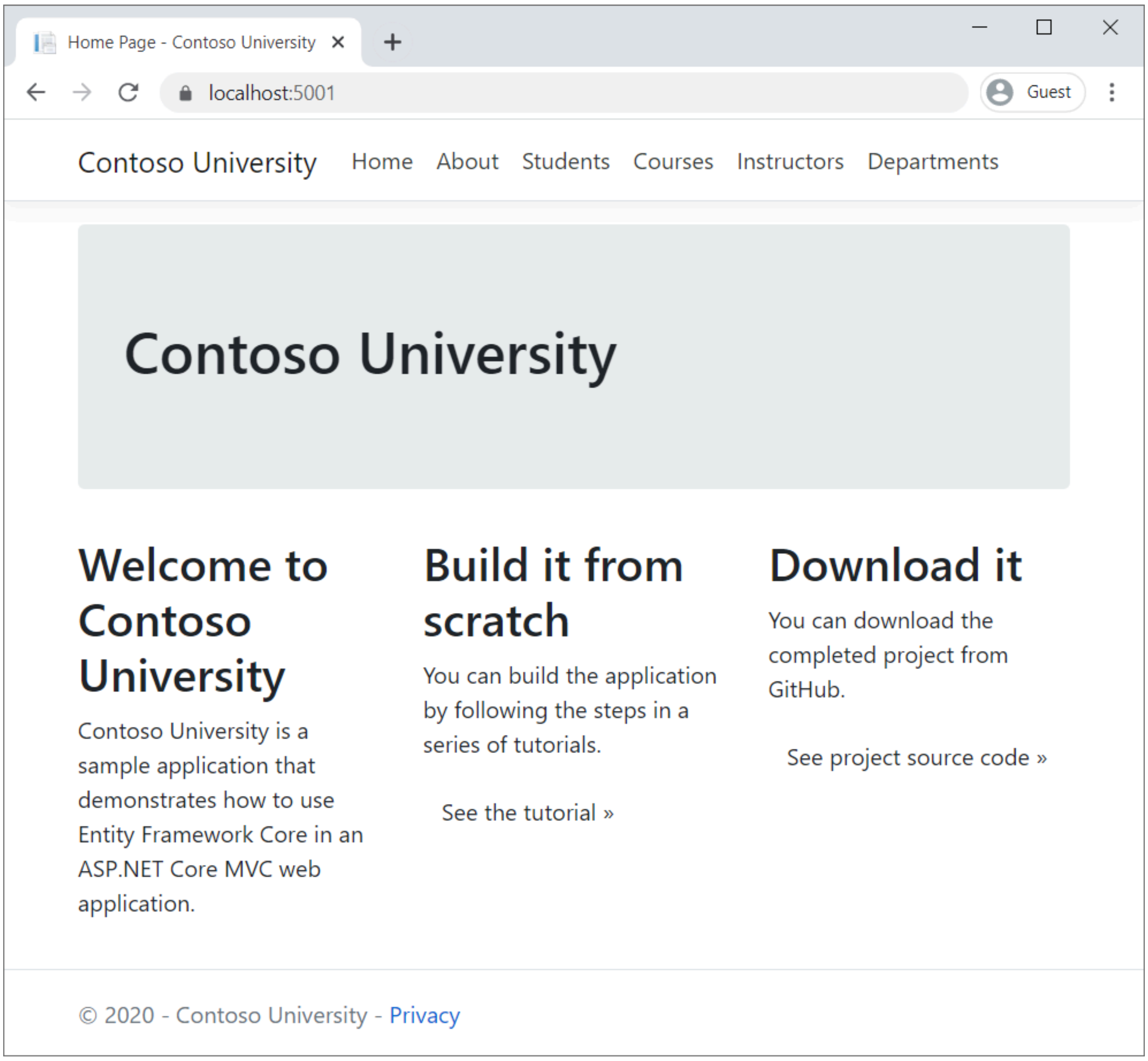
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" 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-controller="Home"
asp-action="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
justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Students" asp-action="Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Courses" asp-action="Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Instructors" asp-action="Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Departments" asp-action="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; 2020 - Contoso University - <a asp-area="" asp-
controller="Home" asp-action="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>
```

In `Views/Home/Index.cshtml`, replace the contents of the file with the following markup:

CSHTML

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

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core MVC web application.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series
of tutorials.</p>
        <p><a class="btn btn-default"
href="https://docs.asp.net/en/latest/data/ef-mvc/intro.html">See the
tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p><a class="btn btn-default"
href="https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/data/ef
-mvc/intro/samples/5cu-final">See project source code &raquo;</a></p>
    </div>
</div>
```

Press CTRL+F5 to run the project or choose **Debug** > **Start Without Debugging** from the menu. The home page is displayed with tabs for the pages created in this tutorial.



Contoso University is a sample application that demonstrates how to use Entity Framework Core in an ASP.NET Core MVC web application.

You can build the application by following the steps in a series of tutorials.

You can download the completed project from GitHub.

EF Core NuGet packages

This tutorial uses SQL Server, and the provider package is Microsoft.EntityFrameworkCore.SqlServer.

The EF SQL Server package and its dependencies, `Microsoft.EntityFrameworkCore` and `Microsoft.EntityFrameworkCore.Relational`, provide runtime support for EF.

Add the Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore NuGet package. In the Package Manager Console (PMC), enter the following commands to add the NuGet packages:

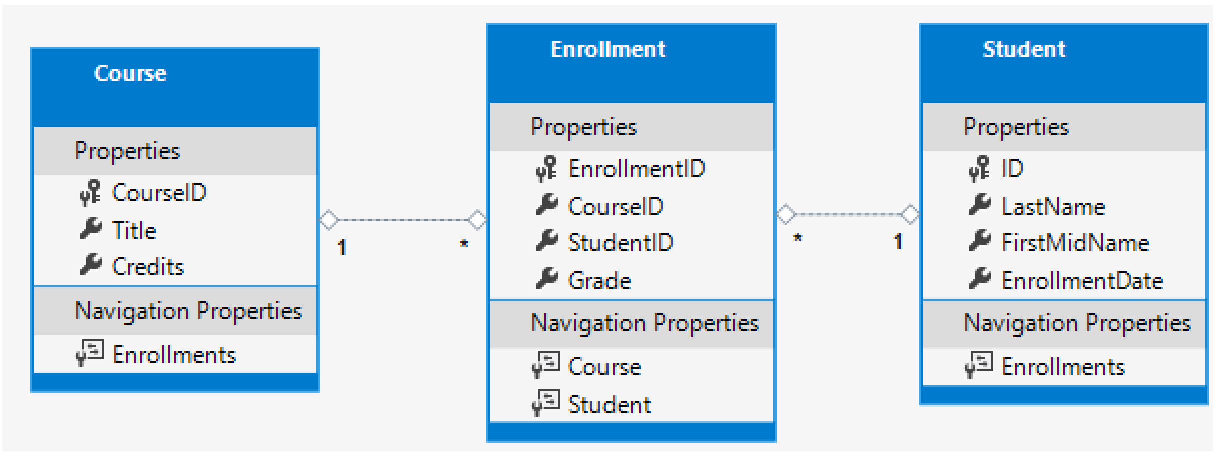
```PowerShell
Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

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

For information about other database providers that are available for EF Core, see Database providers.

Create the data model

The following entity classes are created for this app:

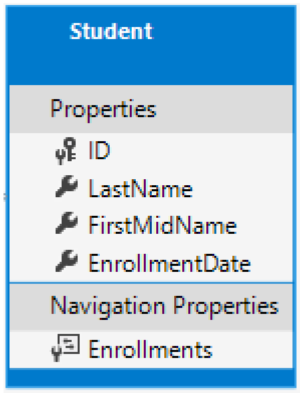


The preceding entities have the following relationships:

- A one-to-many relationship between `Student` and `Enrollment` entities. A student can be enrolled in any number of courses.
- A one-to-many relationship between `Course` and `Enrollment` entities. A course can have any number of students enrolled in it.

In the following sections, a class is created for each of these entities.

The Student entity



In the *Models* folder, create the `Student` class with the following code:

```csharp
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 is the primary key (**PK**) column of the database table that corresponds to this class. By default, EF interprets a property that's named `ID` or `classnameID` as the primary key. For example, the PK could be named `StudentID` rather than `ID`.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. The `Enrollments` property of a `Student` entity:

- Contains all of the `Enrollment` entities that are related to that `Student` entity.
- If a specific `Student` row in the database has two related `Enrollment` rows:
    - That `Student` entity's `Enrollments` navigation property contains those two `Enrollment` entities.

`Enrollment` rows contain a student's PK value in the `StudentID` foreign key (**FK**) column.

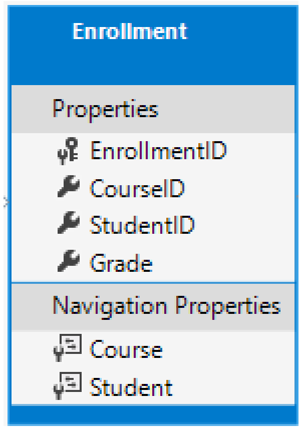
If a navigation property can hold multiple entities:

- The type must be a list, such as `ICollection<T>`, `List<T>`, or `HashSet<T>`.

- Entities can be added, deleted, and updated.

Many-to-many and one-to-many navigation relationships can contain multiple entities. When `ICollection<T>` is used, EF creates a `HashSet<T>` collection by default.

The Enrollment entity



In the *Models* folder, create the `Enrollment` class with the following code:

C#

```csharp
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 PK. This entity uses the `classnameID` pattern instead of `ID` by itself. The `Student` entity used the `ID` pattern. Some developers prefer to use one pattern throughout the data model. In this tutorial, the variation illustrates that either pattern can be used. A later tutorial shows how using `ID` without classname makes it easier to implement inheritance in the data model.

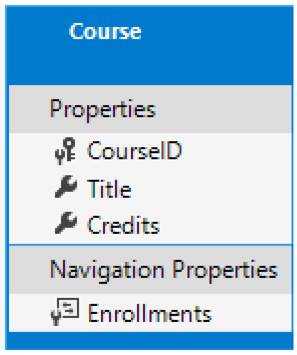
The `Grade` property is an `enum`. The `?` after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's `null` is different from a zero grade. `null` means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key (FK), and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property can only hold a single `Student` entity. This differs from the `Student.Enrollments` navigation property, which can hold multiple `Enrollment` entities.

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

Entity Framework interprets a property as a FK property if it's named `<`navigation property name`><`primary key property name`>`. For example, `StudentID` for the `Student` navigation property since the `Student` entity's PK is `ID`. FK properties can also be named `<`primary key property name`>`. For example, `CourseID` because the `Course` entity's PK is `CourseID`.

The Course entity



In the *Models* folder, create the `Course` class with the following code:

```csharp
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 is explained in a later tutorial. This attribute allows entering the PK for the course rather than having the database generate it.

Create the database context

The main class that coordinates EF functionality for a given data model is the DbContext database context class. This class is created by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class. The `DbContext` derived class specifies which entities are included in the data model. Some EF behaviors can be customized. In this project, the class is named `SchoolContext`.

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

In the *Data* folder create a `SchoolContext` class with the following code:

C#

```csharp
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; }
    }
}
```

The preceding code creates a `DbSet` property for each entity set. In EF terminology:

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

The `DbSet<Enrollment>` and `DbSet<Course>` statements could be omitted and it would work the same. EF would include them implicitly because:

- The `Student` entity references the `Enrollment` entity.
- The `Enrollment` entity references the `Course` entity.

When the database is created, EF creates tables that have names the same as the `DbSet` property names. Property names for collections are typically plural. For example, `Students` rather than `Student`. Developers disagree about whether table names should be pluralized or not. For these tutorials, the default behavior is overridden by specifying singular table names in the `DbContext`. To do that, add the following highlighted code after the last DbSet property.

C#

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

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

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

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

Register the `SchoolContext`

ASP.NET Core includes dependency injection. Services, such as the EF database context, are registered with dependency injection during app startup. Components that require these services, such as MVC controllers, are provided these services via constructor parameters. The controller constructor code that gets a context instance is shown later in this tutorial.

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

C#

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

namespace ContosoUniversity
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<SchoolContext>(options =>

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

            services.AddControllersWithViews();
        }
```

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

Open the `appsettings.json` file and add a connection string as shown in the following markup:

JSON

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Trusted_Connection=True;
MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Add the database exception filter

Add AddDatabaseDeveloperPageExceptionFilter to `ConfigureServices` as shown in the following code:

```C#
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<SchoolContext>(options =>

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

    services.AddDatabaseDeveloperPageExceptionFilter();

    services.AddControllersWithViews();
}
```

The `AddDatabaseDeveloperPageExceptionFilter` provides helpful error information in the development environment.

SQL Server Express LocalDB

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

Initialize DB with test data

EF creates an empty database. In this section, a method is added that's called after the database is created in order to populate it with test data.

The `EnsureCreated` method is used to automatically create the database. In a later tutorial, you see how to handle model changes by using Code First Migrations to change the database schema instead of dropping and re-creating the database.

In the *Data* folder, create a new class named `DbInitializer` with the following code:

```C#
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.P
arse("2005-09-01")},
            new
Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Pa
rse("2002-09-01")},
            new
Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse
("2003-09-01")},
            new
Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Pa
rse("2002-09-01")},
            new
Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002
-09-01")},
            new
Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Pars
e("2001-09-01")},
            new
Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse
("2003-09-01")},
            new
Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Pars
e("2005-09-01")}
            };
            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();

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

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

The preceding code checks if the database exists:

- If the database is not found;
  - It is created and loaded with test data. It loads test data into arrays rather than `List<T>` collections to optimize performance.
- If the database is found, it takes no action.

Update `Program.cs` with the following code:

```
C#

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>
();
                    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>();
                });
    }
}
```

`Program.cs` does the following on app startup:

- Get a database context instance from the dependency injection container.
- Call the `DbInitializer.Initialize` method.
- Dispose the context when the `Initialize` method completes as shown in the following code:

C#

```csharp
public static void Main(string[] args)
{
     var host = CreateWebHostBuilder(args).Build();

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            var context = services.GetRequiredService<SchoolContext>();
            DbInitializer.Initialize(context);
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred while seeding the 
database.");
        }
    }

    host.Run();
}
```

The first time the app is run, the database is created and loaded with test data. Whenever the data model changes:

- Delete the database.
- Update the seed method, and start afresh with a new database.

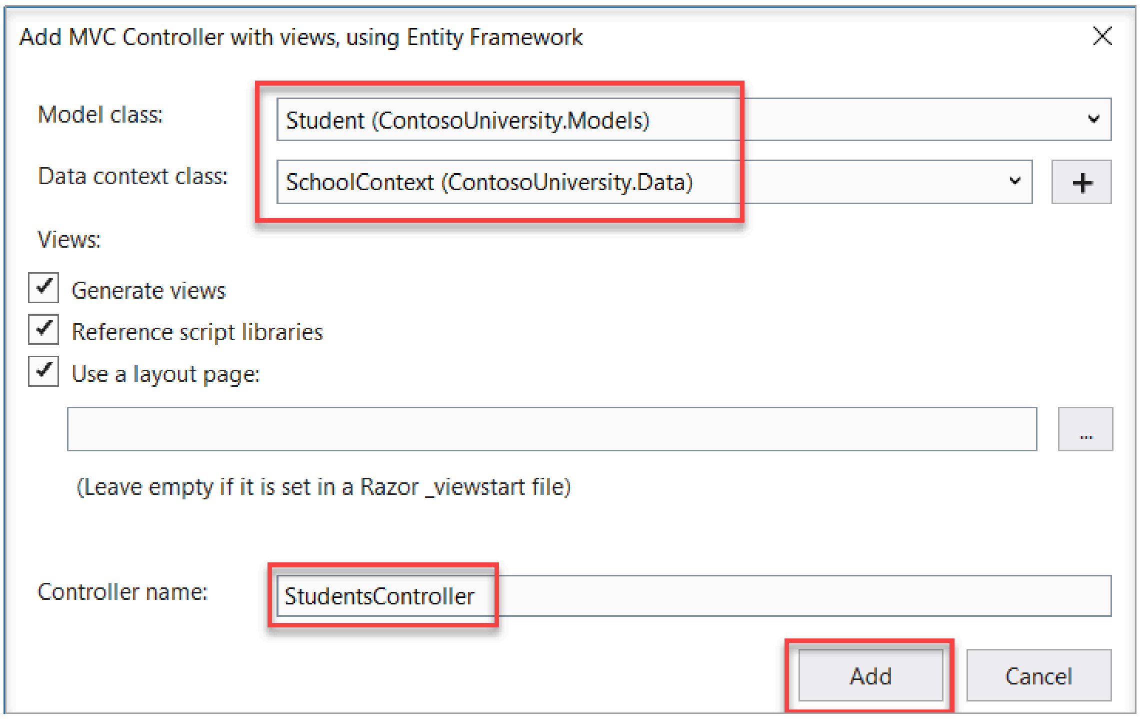
In later tutorials, the database is modified when the data model changes, without deleting and re-creating it. No data is lost when the data model changes.

Create controller and views

Use the scaffolding engine in Visual Studio to add an MVC controller and views that will use EF to query and save data.

The automatic creation of CRUD action methods and views is known as scaffolding.

- In **Solution Explorer**, right-click the `Controllers` folder and select **Add** > **New Scaffolded Item**.
- In the **Add Scaffold** dialog box:
  - Select **MVC controller with views, using Entity Framework**.
  - Click **Add**. The **Add MVC Controller with views, using Entity Framework** dialog box appears:



Add MVC Controller with views, using Entity Framework

Model class: Student (ContosoUniversity.Models)

Data context class: SchoolContext (ContosoUniversity.Data)

Views:

✓ Generate views

✓ Reference script libraries

✓ Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

Controller name: StudentsController

  - In **Model class**, select **Student**.
  - In **Data context class**, select **SchoolContext**.
  - Accept the default **StudentsController** as the name.
  - Click **Add**.

The Visual Studio scaffolding engine creates a `StudentsController.cs` file and a set of views (`*.cshtml` files) that work with the controller.

Notice the controller takes a `SchoolContext` as a constructor parameter.

C#

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

        public StudentsController(SchoolContext context)
        {
            _context = context;
        }
```

ASP.NET Core dependency injection takes care of passing an instance of `SchoolContext` into the controller. You configured that in the `Startup` class.

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

C#

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

The asynchronous programming elements in this code are explained later in the tutorial.

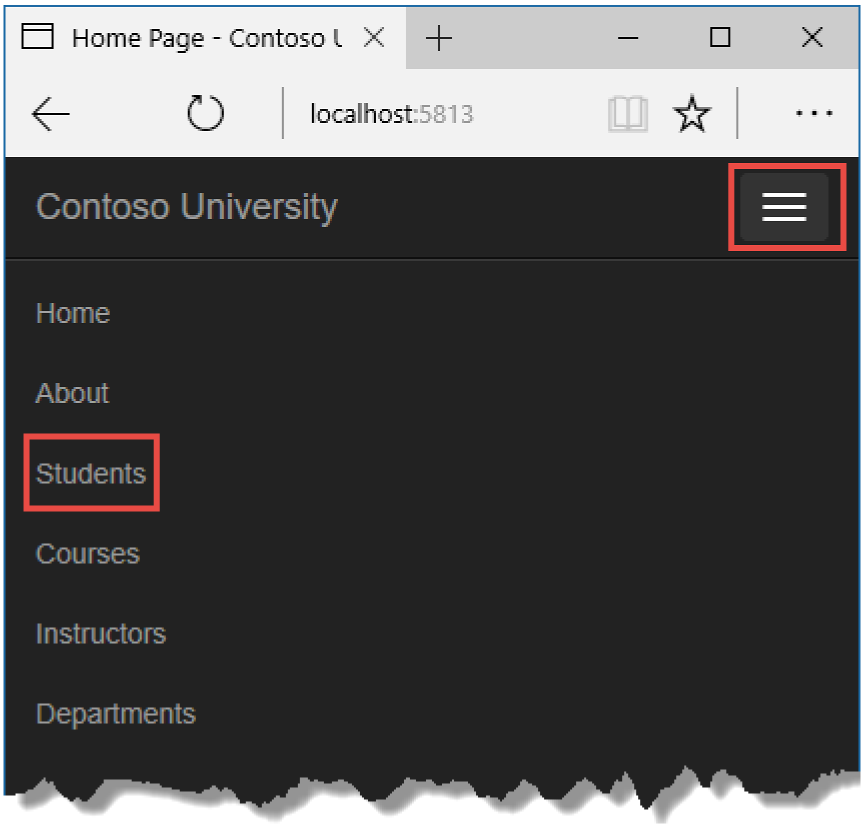
The `Views/Students/Index.cshtml` view displays this list in a table:

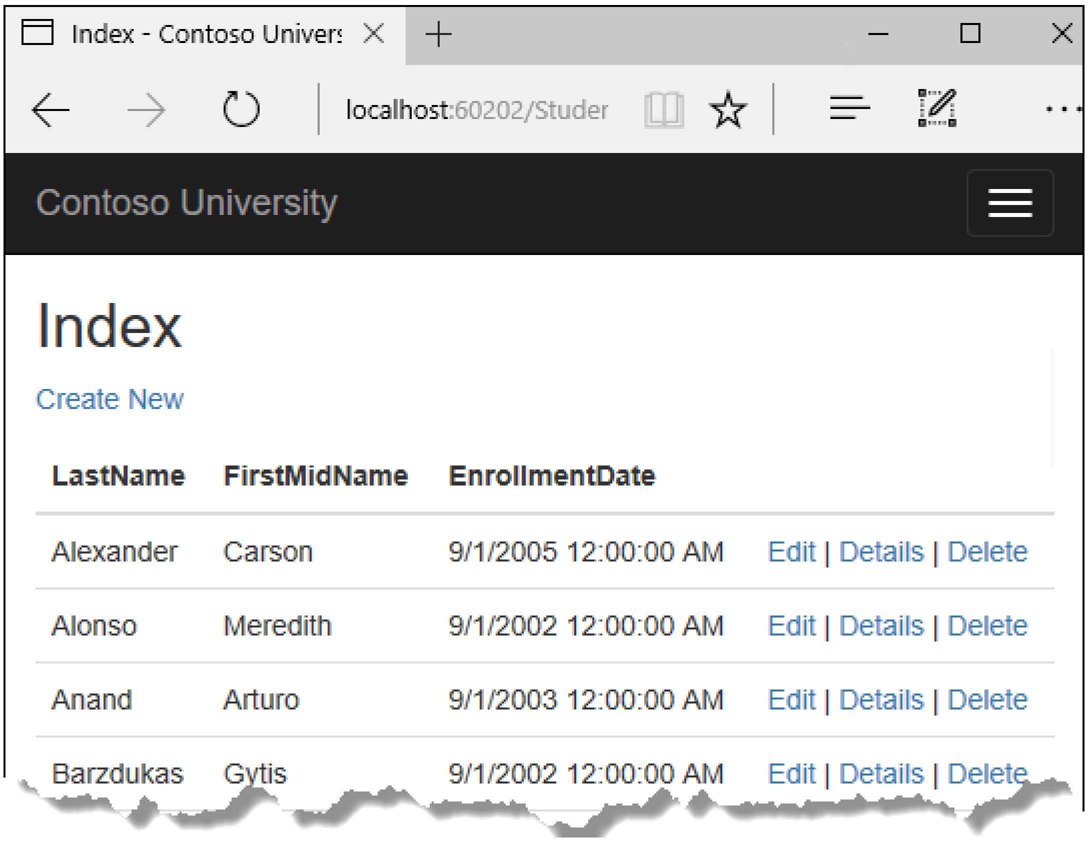
CSHTML

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

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

<h2>Index</h2>

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

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

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





Index

Create New

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

View the database

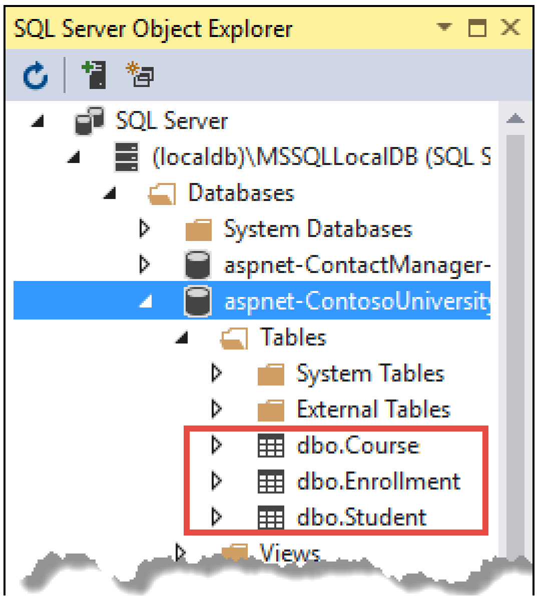
When the app is started, the `DbInitializer.Initialize` method calls `EnsureCreated`. EF saw that there was no database:

- So it created a database.

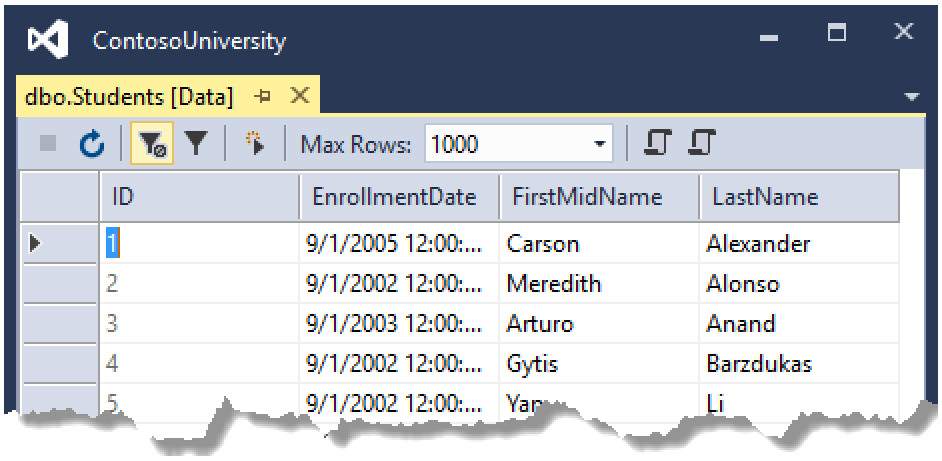
- The `Initialize` method code populated the database with data.

Use **SQL Server Object Explorer** (SSOX) to view the database in Visual Studio:

- Select **SQL Server Object Explorer** from the **View** menu in Visual Studio.
- In SSOX, select **(localdb)\MSSQLLocalDB > Databases**.
- Select `ContosoUniversity1`, the entry for the database name that's in the connection string in the `appsettings.json` file.
- Expand the **Tables** node to see the tables in the database.



Right-click the **Student** table and click **View Data** to see the data in the table.



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

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

Because `EnsureCreated` is called in the initializer method that runs on app start, you could:

- Make a change to the `Student` class.
- Delete the database.

- Stop, then start the app. The database is automatically re-created to match the change.

For example, if an `EmailAddress` property is added to the `Student` class, a new `EmailAddress` column in the re-created table. The view won't display the new `EmailAddress` property.

Conventions

The amount of code written in order for the EF to create a complete database is minimal because of the use of the conventions EF uses:

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

Conventional behavior can be overridden. For example, table names can be explicitly specified, as shown earlier in this tutorial. Column names and any property can be set as a PK or FK.

Asynchronous code

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

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

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

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

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

- The `async` keyword tells the compiler to generate callbacks for parts of the method body and to automatically create the `Task<IActionResult>` object that's returned.
- The return type `Task<IActionResult>` represents ongoing work with a result of type `IActionResult`.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that'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:

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

For more information about asynchronous programming in .NET, see Async Overview.

Limit entities fetched

See Performance considerations for information on limiting the number of entities returned from a query.

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:

JSON

```json
{
  "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.

Advance to the next tutorial to learn how to perform basic CRUD (create, read, update, delete) operations.

Implement basic CRUD functionality

Tutorial: Implement CRUD Functionality - ASP.NET MVC with EF Core

Article • 04/10/2024

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

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

In this tutorial, you:

- ✓ Customize the Details page
- ✓ Update the Create page
- ✓ Update the Edit page
- ✓ Update the Delete page
- ✓ Close database connections

Prerequisites

- Get started with EF Core and ASP.NET Core MVC

Customize the Details page

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

In `Controllers/StudentsController.cs`, the action method for the Details view uses the `FirstOrDefaultAsync` method to retrieve a single `Student` entity. Add code that calls

`Include`. `ThenInclude`, and `AsNoTracking` methods, as shown in the following highlighted code.

```
C#
```

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

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

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

    return View(student);
}
```

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

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

Route data

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

```
C#
```

```csharp
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

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

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

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

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

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

HTML

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

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

HTML

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

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

HTML

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

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

HTML

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

For more information about tag helpers, see Tag Helpers in ASP.NET Core.

Add enrollments to the Details view

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

CSHTML

```
<dt class="col-sm-2">
    @Html.DisplayNameFor(model => model.LastName)
</dt>
<dd class="col-sm-10">
    @Html.DisplayFor(model => model.LastName)
</dd>
```

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

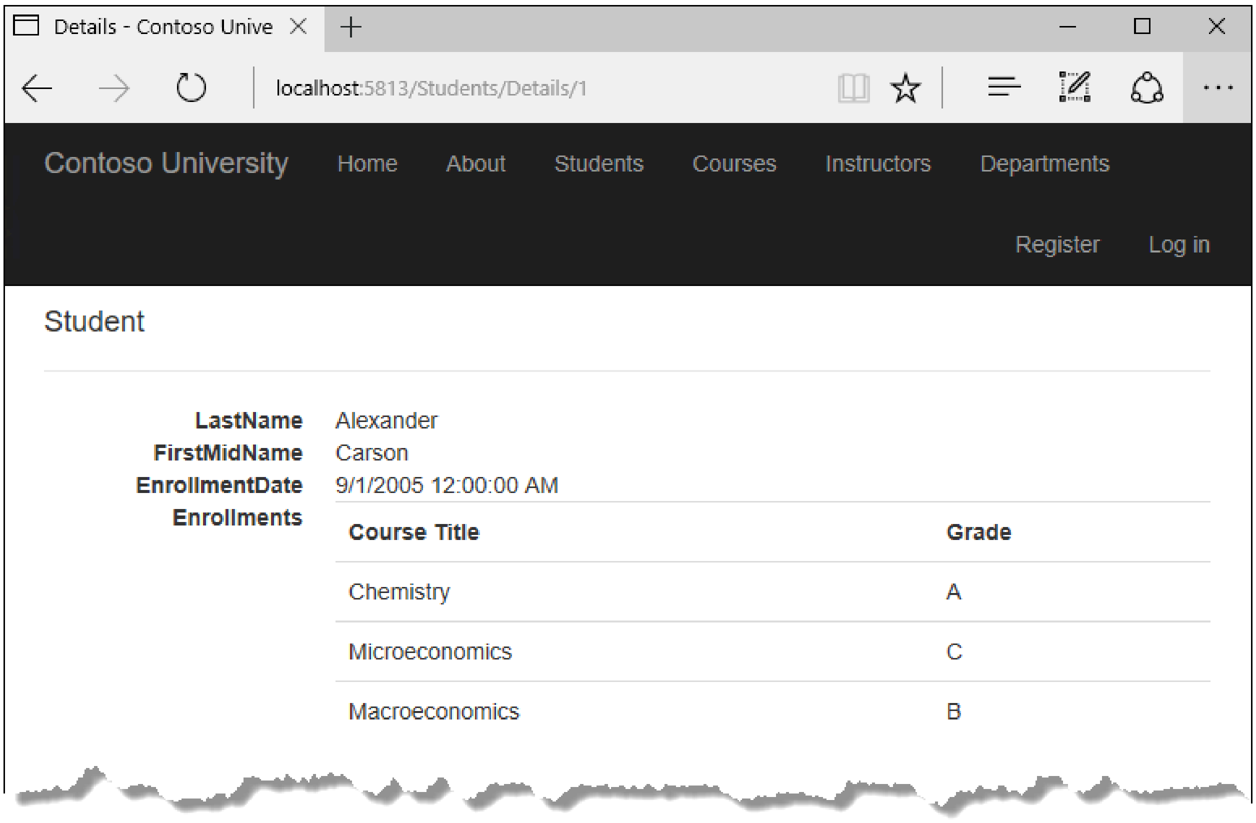
CSHTML

```
<dt class="col-sm-2">
    @Html.DisplayNameFor(model => model.Enrollments)
</dt>
<dd class="col-sm-10">
    <table class="table">
        <tr>
            <th>Course Title</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Course.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
</dd>
```

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

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

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



Student

LastName Alexander
FirstMidName Carson
EnrollmentDate 9/1/2005 12:00:00 AM
Enrollments

Course Title	Grade
Chemistry	A
Microeconomics	C
Macroeconomics	B

Update the Create page

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

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

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

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

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

The `ValidateAntiForgeryToken` attribute helps prevent cross-site request forgery (CSRF) attacks. The token is automatically injected into the view by the FormTagHelper and is included when the form is submitted by the user. The token is validated by the `ValidateAntiForgeryToken` attribute. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

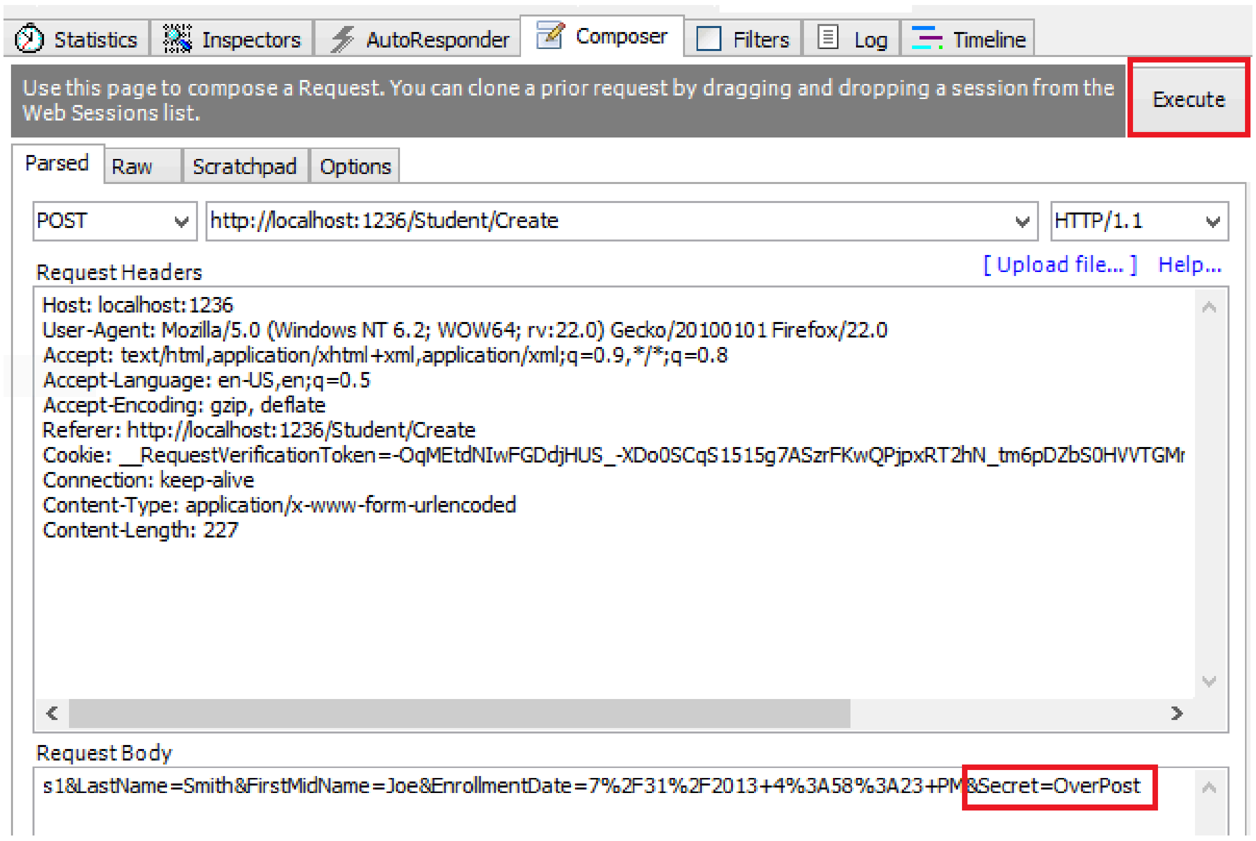
Security note about overposting

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

C#

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

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



Use this page to compose a Request. You can clone a prior request by dragging and dropping a session from the Web Sessions list.

POST http://localhost:1236/Student/Create HTTP/1.1

Request Headers

```
Host: localhost:1236
User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64; rv:22.0) Gecko/20100101 Firefox/22.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost:1236/Student/Create
Cookie: __RequestVerificationToken=-OqMEtdNIwFGDdjHUS_-XDo0SCqS1515g7ASzrFKwQPjpxRT2hN_tm6pDZbS0HVVTGMr
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 227
```

Request Body

```
s1&LastName=Smith&FirstMidName=Joe&EnrollmentDate=7%2F31%2F2013+4%3A58%3A23+PM&Secret=OverPost
```

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

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

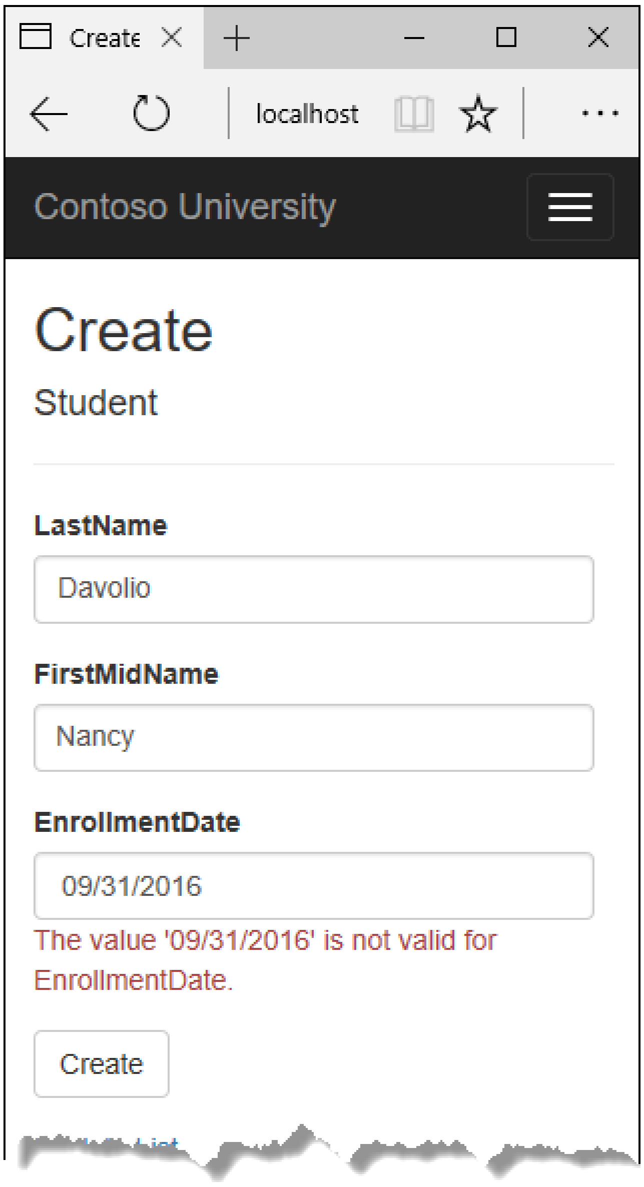
An alternative way to prevent overposting that's preferred by many developers is to use view models rather than entity classes with model binding. Include only the properties you want to update in the view model. Once the MVC model binder has finished, copy the view model properties to the entity instance,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

Test the Create page

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

Run the app, select the **Students** tab, and click **Create New**.

Enter names and a date. Try entering an invalid date if your browser lets you do that. (Some browsers force you to use a date picker.) Then click **Create** to see the error message.



Create

Student

LastName

Davolio

FirstMidName

Nancy

EnrollmentDate

09/31/2016

The value '09/31/2016' is not valid for EnrollmentDate.

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

C#

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

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

Update the Edit page

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

Recommended HttpPost Edit code: Read and update

Replace the HttpPost Edit action method with the following code.

C#

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

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

The new code reads the existing entity and calls `TryUpdateModel` to update fields in the retrieved entity based on user input in the posted form data. The Entity Framework's automatic change tracking sets the `Modified` flag on the fields that are changed by form input. When the `SaveChanges` method is called, the Entity Framework creates SQL statements to update the database row. Concurrency conflicts are ignored, and only the table columns that were updated by the user are updated in the database. (A later tutorial shows how to handle concurrency conflicts.)

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

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

Alternative HttpPost Edit code: Create and attach

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

C#

```
public async Task<IActionResult> Edit(int id,
[Bind("ID,EnrollmentDate,FirstMidName,LastName")] Student student)
```

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

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

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

Entity States

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

An entity may be in one of the following states:

- `Added`. The entity doesn't yet exist in the database. The `SaveChanges` method issues an INSERT statement.
- `Unchanged`. Nothing needs to be done with this entity by the `SaveChanges` method. When you read an entity from the database, the entity starts out with this status.

- `Modified`. Some or all of the entity's property values have been modified. The `SaveChanges` method issues an UPDATE statement.
- `Deleted`. The entity has been marked for deletion. The `SaveChanges` method issues a DELETE statement.
- `Detached`. The entity isn't being tracked by the database context.

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

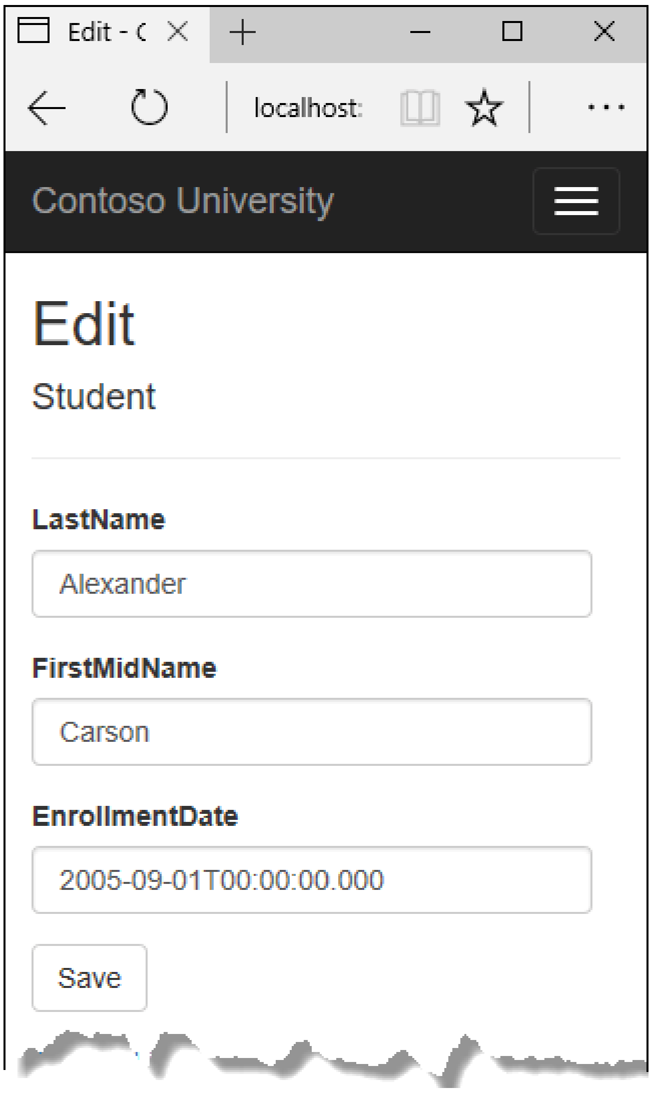
In a web app, the `DbContext` that initially reads an entity and displays its data to be edited is disposed after a page is rendered. When the HttpPost `Edit` action method is called, a new web request is made and you have a new instance of the `DbContext`. If you re-read the entity in that new context,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

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

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

Test the Edit page

Run the app, select the **Students** tab, then click an **Edit** hyperlink.



Edit

Student

LastName

Alexander

FirstMidName

Carson

EnrollmentDate

2005-09-01T00:00:00.000

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

Update the Delete page

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

As you saw for update and create operations, delete operations require two action methods. The method that's called in response to a GET request displays a view that gives the user a chance to approve or cancel the delete operation. If the user approves it, a POST request is created. When that happens, the HttpPost `Delete` method is called and then that method actually performs the delete operation.

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

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

C#

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

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

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

    return View(student);
}
```

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

The read-first approach to HttpPost Delete

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

C#

```csharp
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var student = await _context.Students.FindAsync(id);
    if (student == null)
    {
        return RedirectToAction(nameof(Index));
    }

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

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

The create-and-attach approach to HttpPost Delete

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

C#

```csharp
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    try
    {
        Student studentToDelete = new Student() { ID = id };
        _context.Entry(studentToDelete).State = EntityState.Deleted;
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id,
saveChangesError = true });
    }
}
```

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

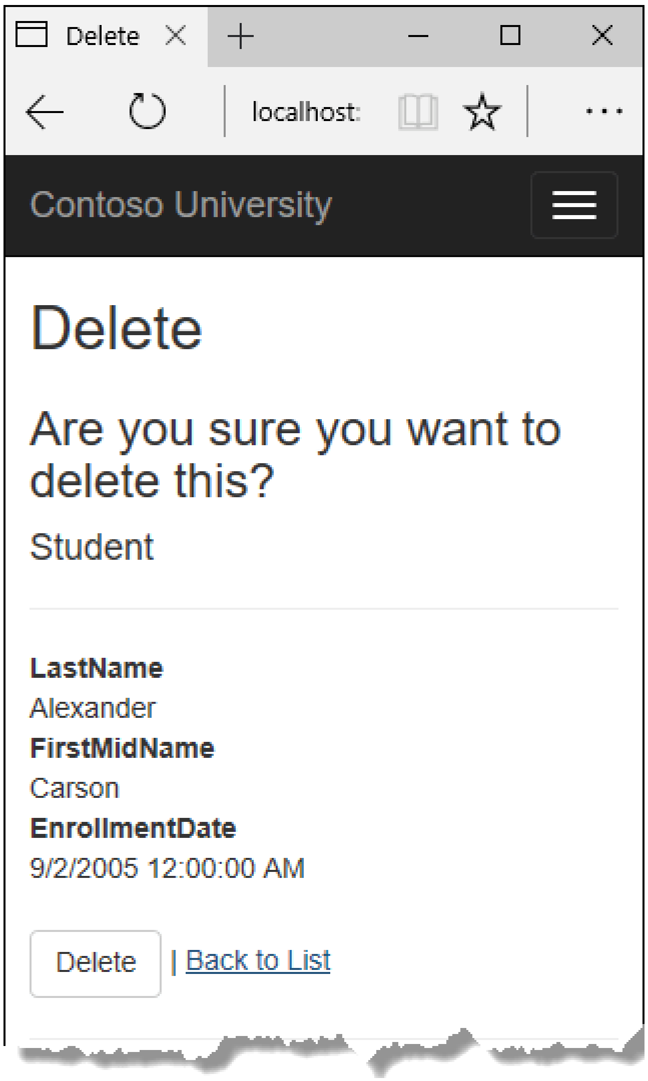
Update the Delete view

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

CSHTML

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

Run the app, select the **Students** tab, and click a **Delete** hyperlink:



Delete

Are you sure you want to delete this?

Student

LastName
Alexander
FirstMidName
Carson
EnrollmentDate
9/2/2005 12:00:00 AM

Delete | Back to List

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

Close database connections

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

In `Startup.cs`, you call the AddDbContext extension method to provision the `DbContext` class in the ASP.NET Core DI container. That method sets the service lifetime to `Scoped` by default. `Scoped` means the context object lifetime coincides with the web request life time, and the `Dispose` method will be called automatically at the end of the web request.

Handle transactions

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

No-tracking queries

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

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

- During the context lifetime you don't need to update any entities, and you don't need EF to automatically load navigation properties with entities retrieved by separate queries. Frequently these conditions are met in a controller's HttpGet action methods.

- You are running a query that retrieves a large volume of data, and only a small portion of the returned data will be updated. It may be more efficient to turn off tracking for the large query, and run a query later for the few entities that need to be updated.

- You want to attach an entity in order to update it, but earlier you retrieved the same entity for a different purpose. Because the entity is already being tracked by the database context, you can't attach the entity that you want to change. One way to handle this situation is to call `AsNoTracking` on the earlier query.

For more information, see Tracking vs. No-Tracking.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Customized the Details page
- ✓ Updated the Create page
- ✓ Updated the Edit page
- ✓ Updated the Delete page
- ✓ Closed database connections

Advance to the next tutorial to learn how to expand the functionality of the **Index** page by adding sorting, filtering, and paging.

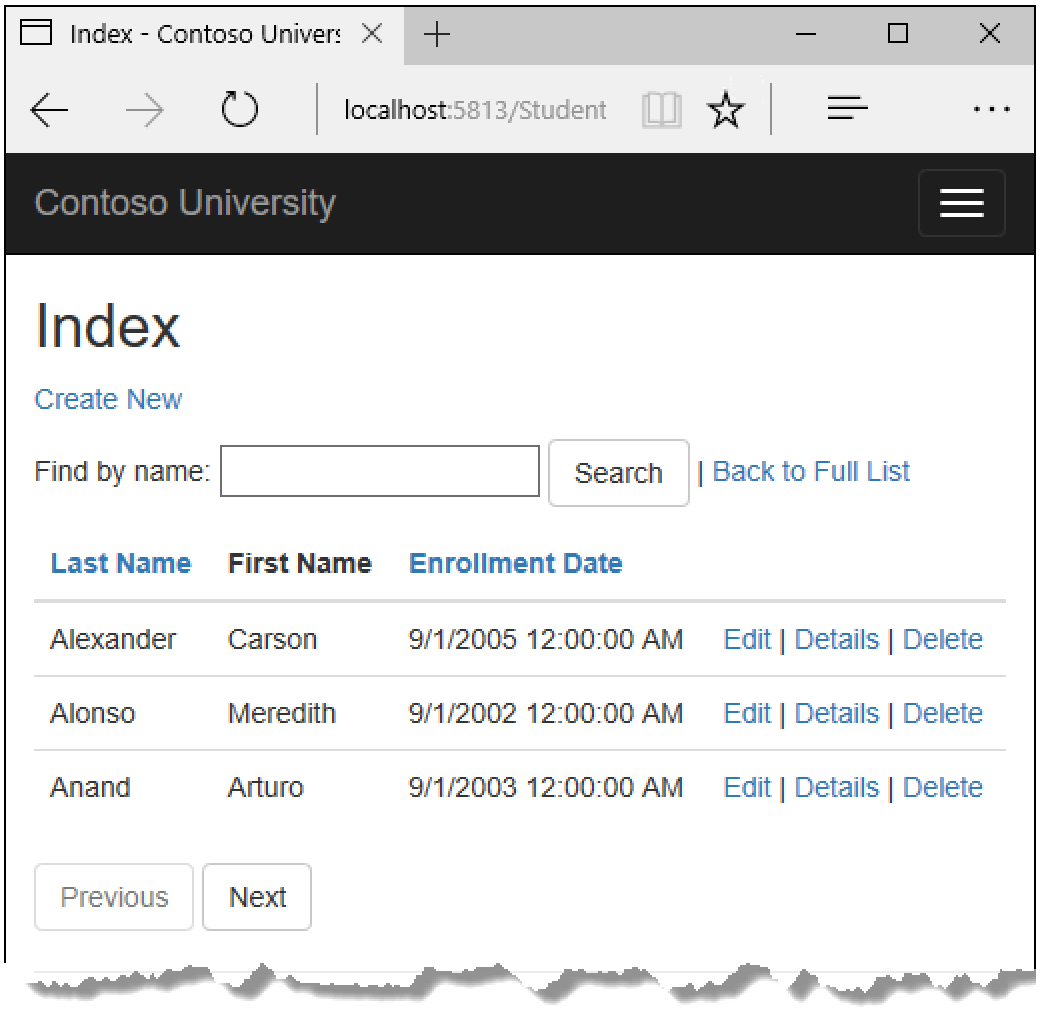
Next: Sorting, filtering, and paging

Tutorial: Add sorting, filtering, and paging - ASP.NET MVC with EF Core

Article • 04/10/2024

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

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



Last Name	First Name	Enrollment Date	
Alexander	Carson	9/1/2005 12:00:00 AM	Edit | Details | Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit | Details | Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit | Details | Delete

In this tutorial, you:

- ✓ Add column sort links
- ✓ Add a Search box
- ✓ Add paging to Students Index
- ✓ Add paging to Index method
- ✓ Add paging links

✓ Create an About page

Prerequisites

- Implement CRUD Functionality

Add column sort links

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

Add sorting Functionality to the Index method

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

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

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

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

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

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

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

Expand table

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

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

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

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

Add column heading hyperlinks to the Student Index view

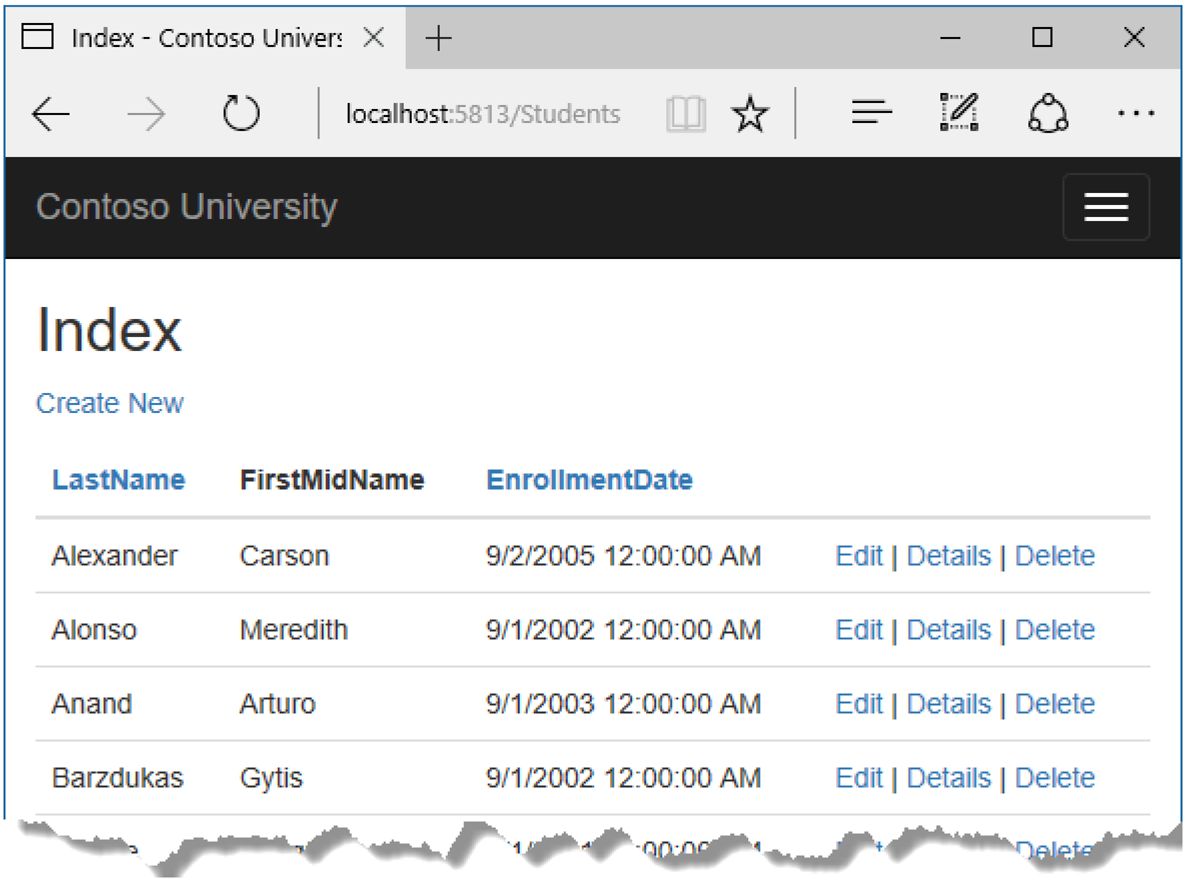
Replace the code in `Views/Students/Index.cshtml`, with the following code to add column heading hyperlinks. The changed lines are highlighted.

CSHTML

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

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

<h2>Index</h2>

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

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

Run the app, select the **Students** tab, and click the **Last Name** and **Enrollment Date** column headings to verify that sorting works.



Index

Create New

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

Add a Search box

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

Add filtering functionality to the Index method

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

C#

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

    var students = from s in _context.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

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

ⓘ **Note**

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

For example, the .NET Framework implementation of the `Contains` method performs a case-sensitive comparison by default, but in SQL Server this is determined by the collation setting of the SQL Server instance. That setting defaults to case-insensitive. You could call the `ToUpper` method to make the test explicitly case-insensitive: *Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())*. That would ensure that results stay the same if you change the code later to use a repository which returns an `IEnumerable` collection instead of an `IQueryable` object. (When you call the `Contains` method on an `IEnumerable` collection, you get the

.NET Framework implementation; when you call it on an `IQueryable` object, you get the database provider implementation.) However, there's a performance penalty for this solution. The `ToUpper` code would put a function in the WHERE clause of the TSQL SELECT statement. That would prevent the optimizer from using an index. Given that SQL is mostly installed as case-insensitive, it's best to avoid the `ToUpper` code until you migrate to a case-sensitive data store.

Add a Search Box to the Student Index View

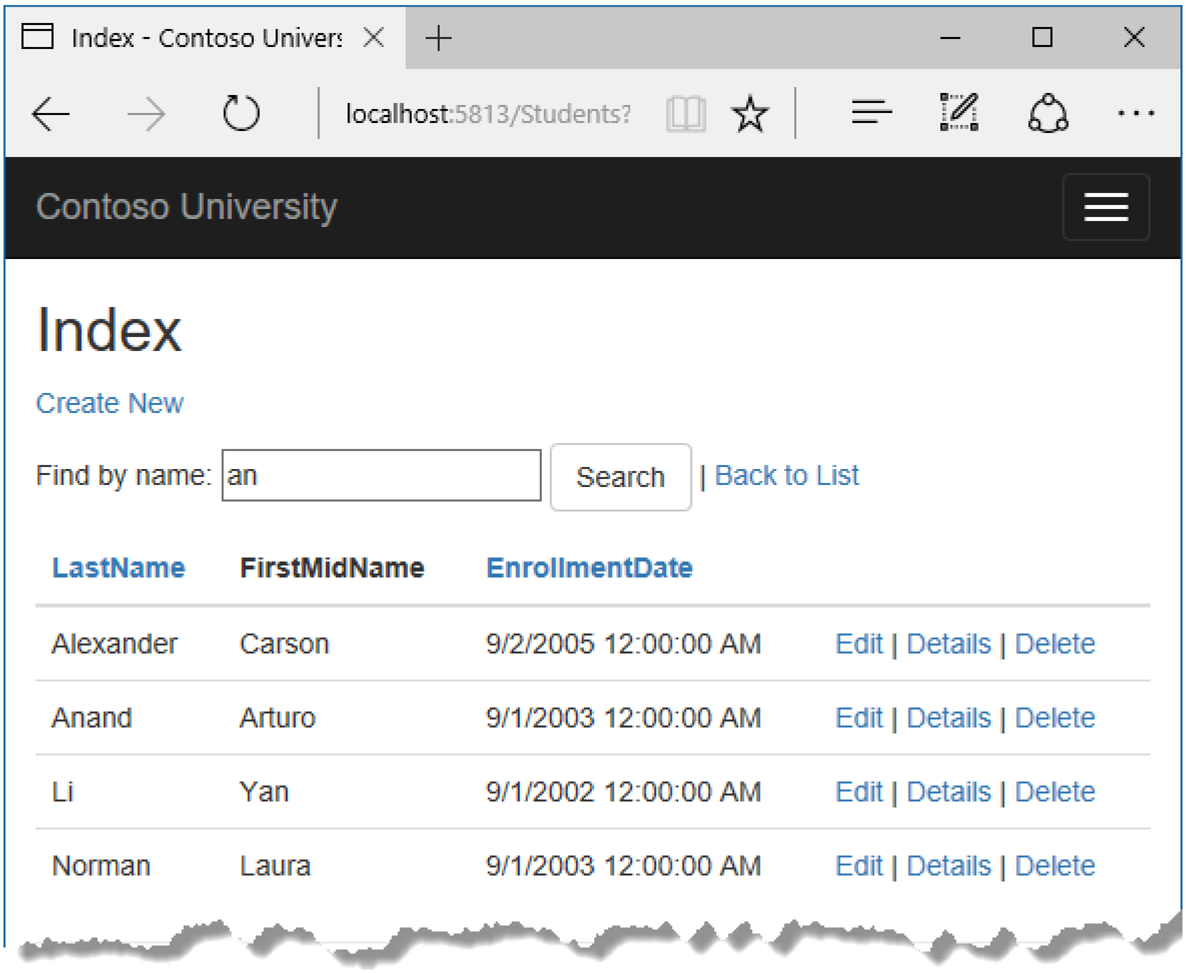
In `Views/Student/Index.cshtml`, add the highlighted code immediately before the opening table tag in order to create a caption, a text box, and a **Search** button.

CSHTML

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

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

<table class="table">
```

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

Run the app, select the **Students** tab, enter a search string, and click Search to verify that filtering is working.


Index - Contoso Univers

localhost:5813/Students?

Contoso University

Index

Create New

Find by name: an Search | Back to List

LastName	FirstMidName	EnrollmentDate	
Alexander	Carson	9/2/2005 12:00:00 AM	Edit | Details | Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit | Details | Delete
Li	Yan	9/1/2002 12:00:00 AM	Edit | Details | Delete
Norman	Laura	9/1/2003 12:00:00 AM	Edit | Details | Delete

Notice that the URL contains the search string.

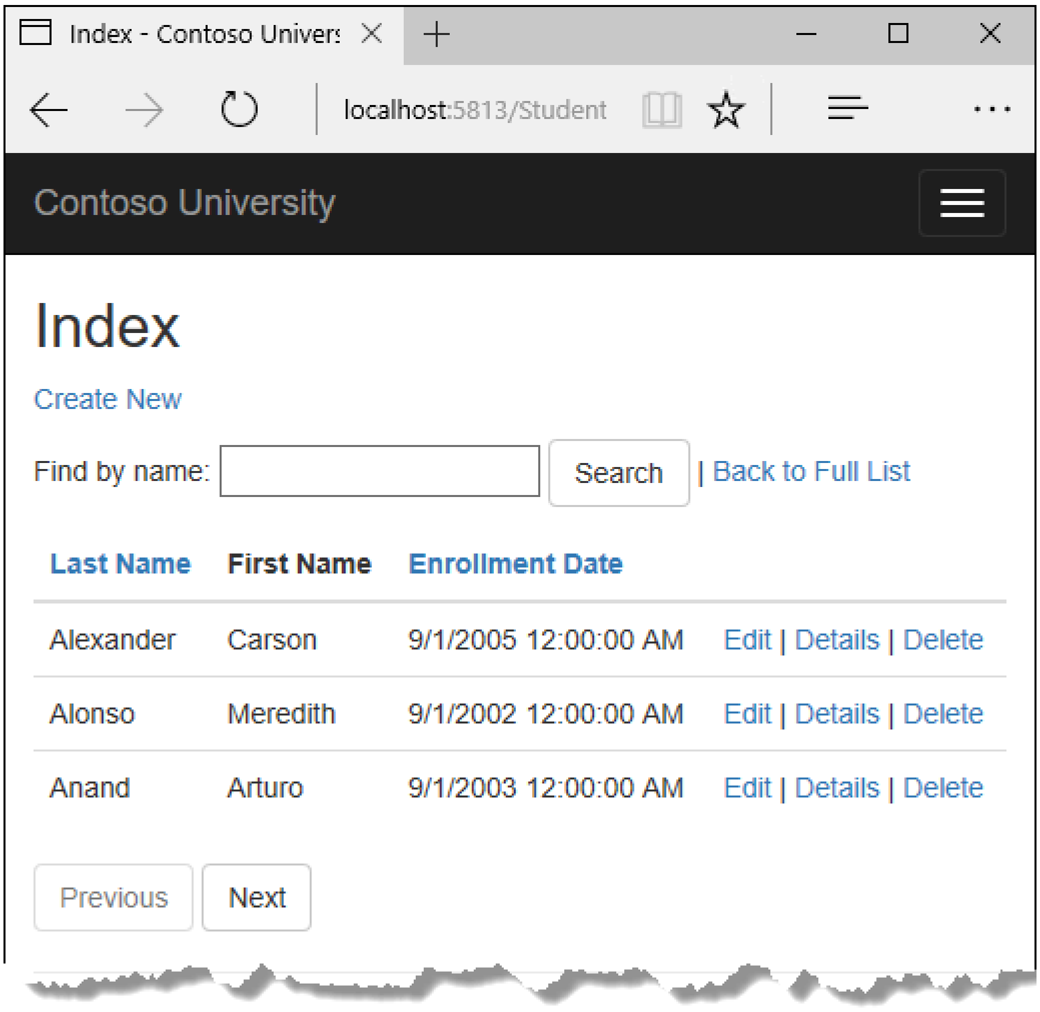
```HTML
http://localhost:5813/Students?SearchString=an
```

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

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

Add paging to Students Index

To add paging to the Students Index page, you'll create a `PaginatedList` class that uses `Skip` and `Take` statements to filter data on the server instead of always retrieving all rows of the table. Then you'll make additional changes in the `Index` method and add paging buttons to the `Index` view. The following illustration shows the paging buttons.



Index

Create New

Find by name: Search | Back to Full List

Last Name	First Name	Enrollment Date	
Alexander	Carson	9/1/2005 12:00:00 AM	Edit | Details | Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit | Details | Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit | Details | Delete

Previous Next

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

C#

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

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

Add paging to Index method

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

C#

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? pageNumber)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc"
: "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    if (searchString != null)
    {
        pageNumber = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewData["CurrentFilter"] = searchString;

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

    int pageSize = 3;
    return View(await
PaginatedList<Student>.CreateAsync(students.AsNoTracking(), pageNumber ?? 1,
pageSize));
}
```

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

C#

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

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

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

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

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

C#

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

At the end of the `Index` method, 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 then passed to the view.

C#

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

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

Add paging links

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

CSHTML

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

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

<h2>Index</h2>

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

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

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

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

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

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

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

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

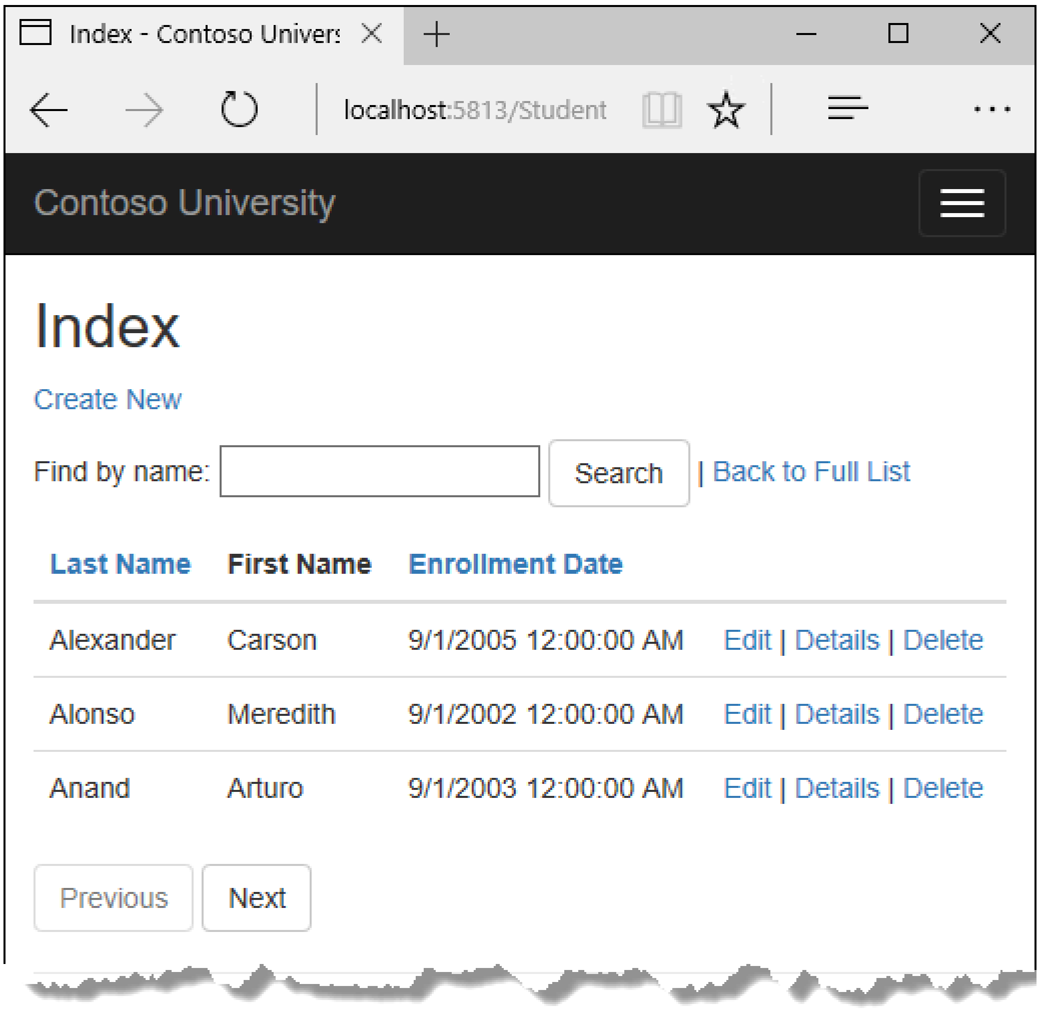
HTML

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

The paging buttons are displayed by tag helpers:

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

Run the app and go to the Students page.



Index

Last Name	First Name	Enrollment Date	
Alexander	Carson	9/1/2005 12:00:00 AM	Edit | Details | Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit | Details | Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit | Details | Delete

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

Create an About page

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

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

Create the view model

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

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

C#

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

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

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

Modify the Home Controller

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

C#

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

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

C#

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    private readonly SchoolContext _context;

    public HomeController(ILogger<HomeController> logger, SchoolContext 
context)
    {
        _logger = logger;
        _context = context;
    }
```

Add an `About` method with the following code:

C#

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

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

Create the About View

Add a `Views/Home/About.cshtml` file with the following code:

CSHTML

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

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

<h2>Student Body Statistics</h2>
```

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

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

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

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Added column sort links
- ✓ Added a Search box
- ✓ Added paging to Students Index
- ✓ Added paging to Index method
- ✓ Added paging links
- ✓ Created an About page

Advance to the next tutorial to learn how to handle data model changes by using migrations.

Next: Handle data model changes

Tutorial: Part 5, apply migrations to the Contoso University sample

Article • 05/31/2024

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

In this tutorial, you:

- ✓ Learn about migrations
- ✓ Create an initial migration
- ✓ Examine Up and Down methods
- ✓ Learn about the data model snapshot
- ✓ Apply the migration

Prerequisites

- Sorting, filtering, and paging

About migrations

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

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

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

Drop the database

Install EF Core tools as a global tool and delete the database:

.NET CLI

```
dotnet tool install --global dotnet-ef
dotnet ef database drop
```

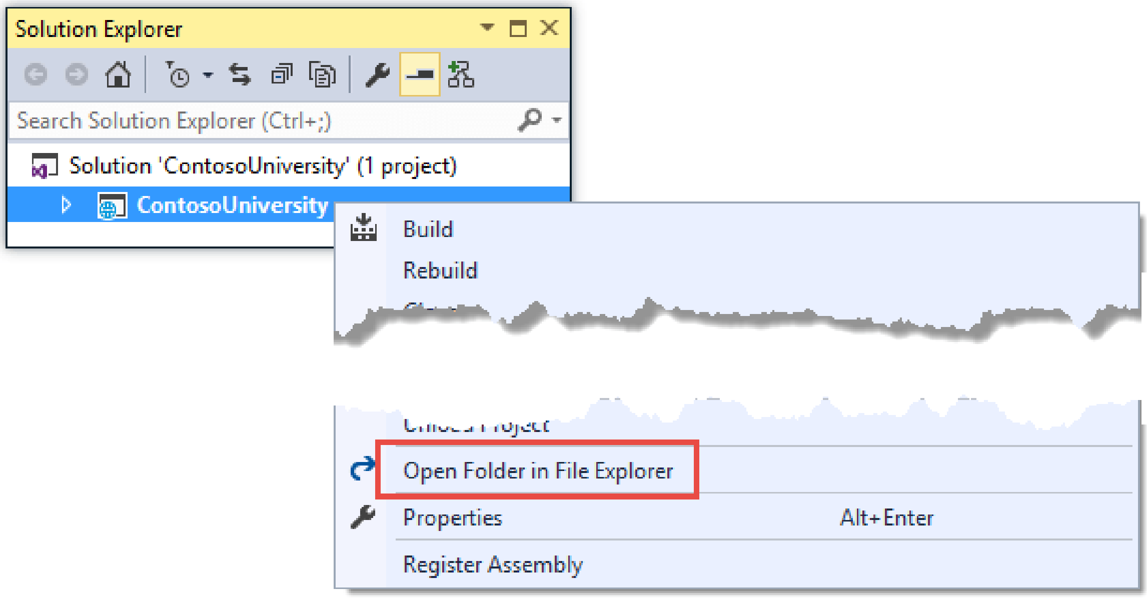
> ⓘ **Note**
>
> By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see dotnet tool install, --arch option. For more information, see GitHub issue dotnet/AspNetCore.Docs #29262.

The following section explains how to run CLI commands.

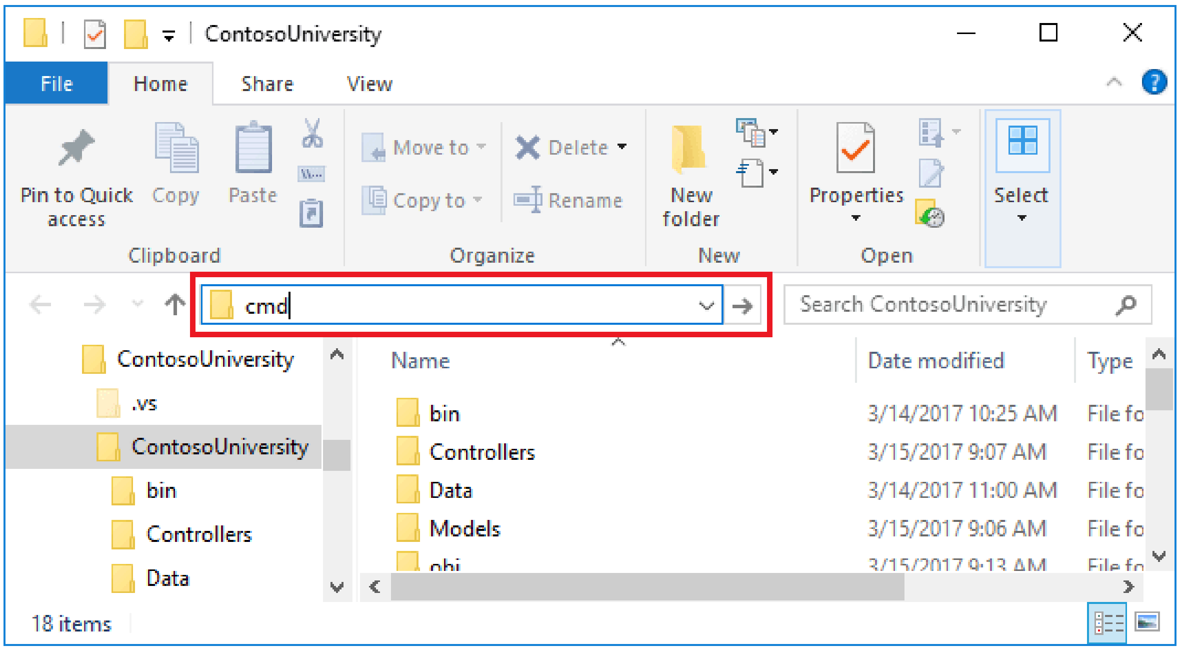
Create an initial migration

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

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


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ContosoUniversity' (1 project)
ContosoUniversity
Build
Rebuild
Open Folder in File Explorer
Properties
Alt+Enter
Register Assembly

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



Name	Date modified	Type
bin	3/14/2017 10:25 AM	File fo
Controllers	3/15/2017 9:07 AM	File fo
Data	3/14/2017 11:00 AM	File fo
Models	3/15/2017 9:06 AM	File fo
obj	3/15/2017 9:13 AM	File fo

Enter the following command in the command window:

.NET CLI

```
dotnet ef migrations add InitialCreate
```

In the preceding commands, output similar to the following is displayed:

Console

```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Done. To undo this action, use 'ef migrations remove'
```

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

Examine Up and Down methods

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

C#

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(nullable: false),
                Credits = table.Column<int>(nullable: false),
                Title = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });

        // Additional code not shown
    }

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

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

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

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

The data model snapshot

Migrations creates 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.

Use the dotnet ef migrations remove command to remove a migration. `dotnet ef migrations remove` deletes the migration and ensures the snapshot is correctly reset. If `dotnet ef migrations remove` fails, use `dotnet ef migrations remove -v` to get more information on the failure.

See EF Core Migrations in Team Environments for more information about how the snapshot file is used.

Apply the migration

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

.NET CLI

```
dotnet ef database update
```

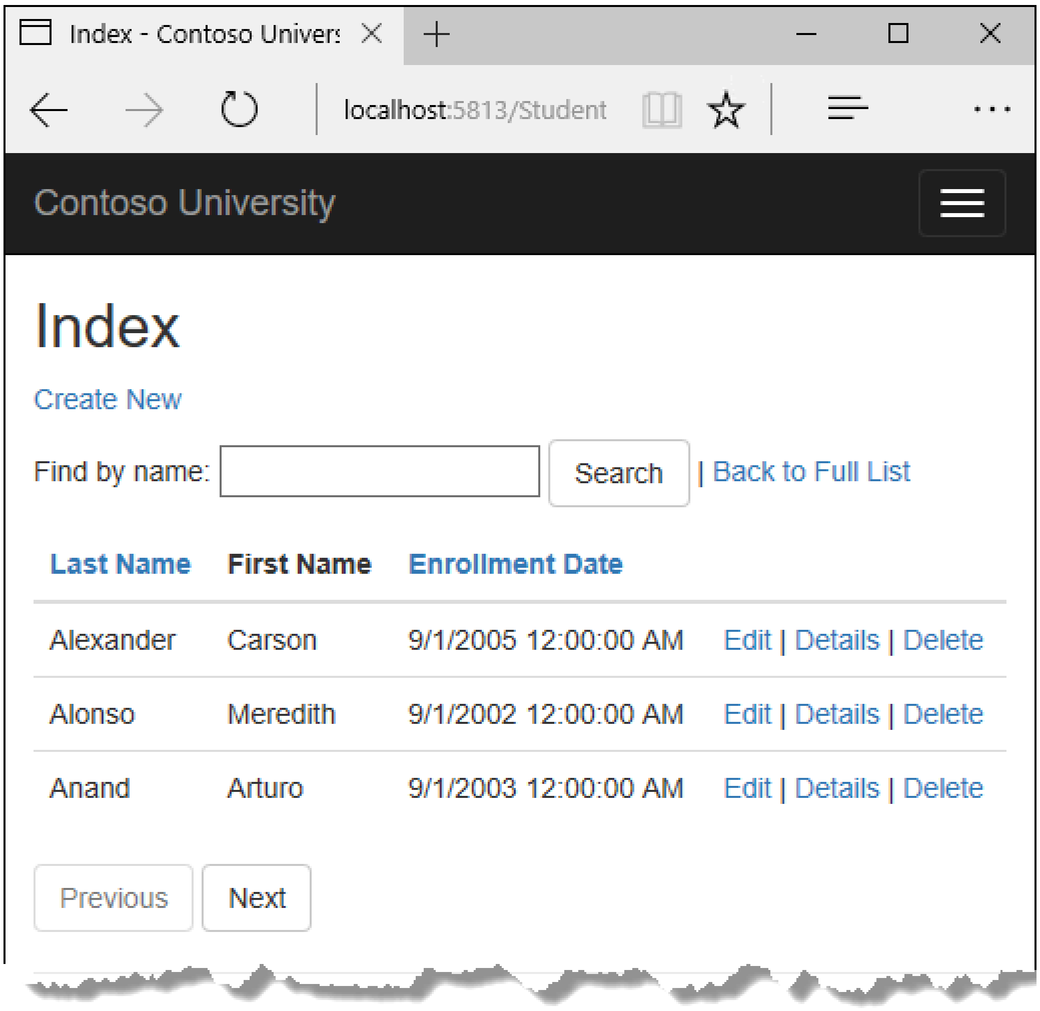
The output from the command is similar to the `migrations add` command, except that you see logs for the SQL commands that set up the database. Most of the logs are omitted in the following sample output. If you prefer not to see this level of detail in log messages, you can change the log level in the `appsettings.Development.json` file. For more information, see Logging in .NET Core and ASP.NET Core.

text

```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (274ms) [Parameters=[], CommandType='Text',
CommandTimeout='60']
      CREATE DATABASE [ContosoUniversity2];
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (60ms) [Parameters=[], CommandType='Text',
CommandTimeout='60']
      IF SERVERPROPERTY('EngineEdition') <> 5
      BEGIN
          ALTER DATABASE [ContosoUniversity2] SET READ_COMMITTED_SNAPSHOT
ON;
      END;
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (15ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
      CREATE TABLE [__EFMigrationsHistory] (
          [MigrationId] nvarchar(150) NOT NULL,
          [ProductVersion] nvarchar(32) NOT NULL,
          CONSTRAINT [PK___EFMigrationsHistory] PRIMARY KEY ([MigrationId])
      );

<logs omitted for brevity>

info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (3ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
      INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
      VALUES (N'20190327172701_InitialCreate', N'5.0-rtm');
Done.
```

Use **SQL Server Object Explorer** to inspect the database as you did in the first tutorial. You'll notice the addition of an __EFMigrationsHistory table that keeps track of which migrations have been applied to the database. View the data in that table and you'll see one row for the first migration. (The last log in the preceding CLI output example shows the INSERT statement that creates this row.)

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



Index

Create New

Find by name: Search | Back to Full List

Last Name	First Name	Enrollment Date	
Alexander	Carson	9/1/2005 12:00:00 AM	Edit | Details | Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit | Details | Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit | Details | Delete

Previous Next

Compare CLI and PMC

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

The EF commands for the PMC commands are in the Microsoft.EntityFrameworkCore.Tools package. This package is included in the Microsoft.AspNetCore.App metapackage, so you don't need to add a package reference if your app has a package reference for `Microsoft.AspNetCore.App`.

Important: This isn't the same package as the one you install for the CLI by editing the `.csproj` file. The name of this one ends in `Tools`, unlike the CLI package name which ends in `Tools.DotNet`.

For more information about the CLI commands, see .NET CLI.

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

Get the code

Download or view the completed application.

Next step

Advance to the next tutorial to begin looking at more advanced topics about expanding the data model. Along the way you'll create and apply additional migrations.

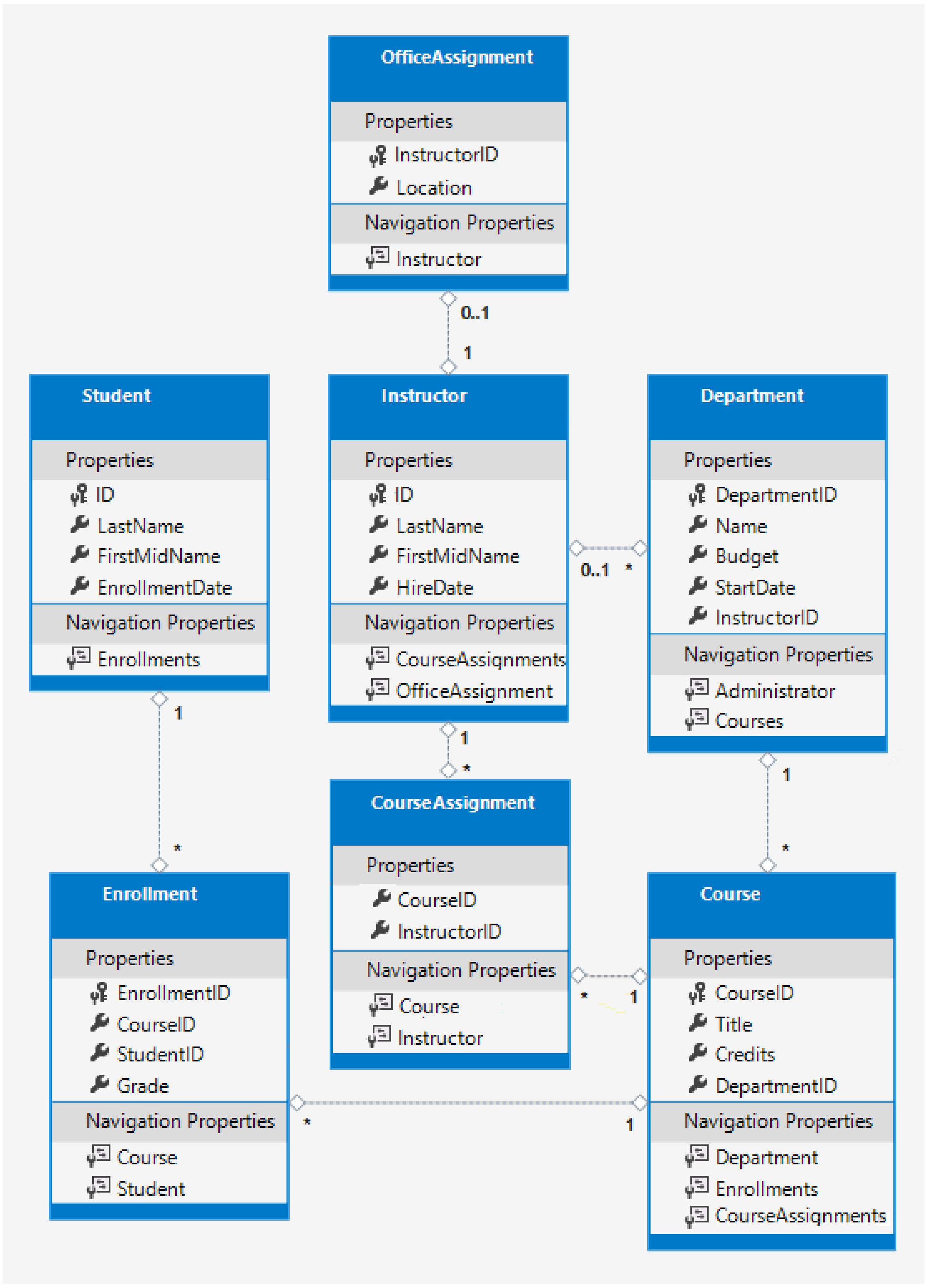
Create and apply additional migrations

Tutorial: Create a complex data model - ASP.NET MVC with EF Core

Article • 04/10/2024

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

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



In this tutorial, you:

- ✓ Customize the Data model
- ✓ Make changes to Student entity
- ✓ Create Instructor entity
- ✓ Create OfficeAssignment entity
- ✓ Modify Course entity
- ✓ Create Department entity
- ✓ Modify Enrollment entity
- ✓ Update the database context

- ✓ Seed database with test data
- ✓ Add a migration
- ✓ Change the connection string
- ✓ Update the database

Prerequisites

- Using EF Core migrations

Customize the Data model

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

The DataType attribute

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

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

C#

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

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

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

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

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

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

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

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

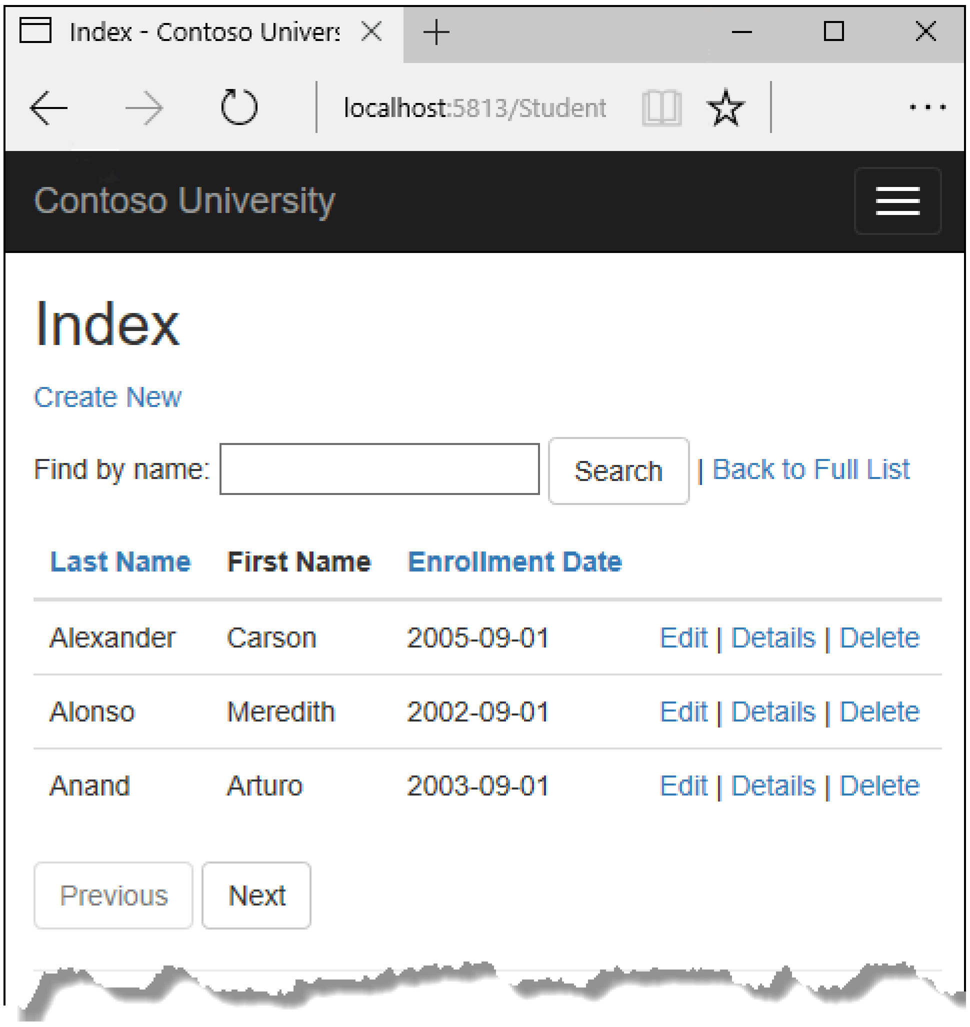
You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute also. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, some client-side input validation, etc.).

- By default, the browser will render data using the correct format based on your locale.

For more information, see the <input> tag helper documentation.

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



Index

Create New

Find by name: Search | Back to Full List

Last Name	First Name	Enrollment Date	
Alexander	Carson	2005-09-01	Edit | Details | Delete
Alonso	Meredith	2002-09-01	Edit | Details | Delete
Anand	Arturo	2003-09-01	Edit | Details | Delete

Previous Next

The StringLength attribute

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

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

C#

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

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

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

C#

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

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

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

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

.NET CLI

```
dotnet ef migrations add MaxLengthOnNames
```

.NET CLI

```
dotnet ef database update
```

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

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

Run the app, select the **Students** tab, click **Create New**, and try to enter either name longer than 50 characters. The application should prevent you from doing this.

The Column attribute

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

The `Column` attribute specifies that when the database is created, the column of the `Student` table that maps to the `FirstMidName` property will be named `FirstName`. In other words, when your code refers to `Student.FirstMidName`, the data will come from or be updated in the `FirstName` column of the `Student` table. If you don't specify column names, they're given the same name as the property name.

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

C#

```csharp
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)]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

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

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

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

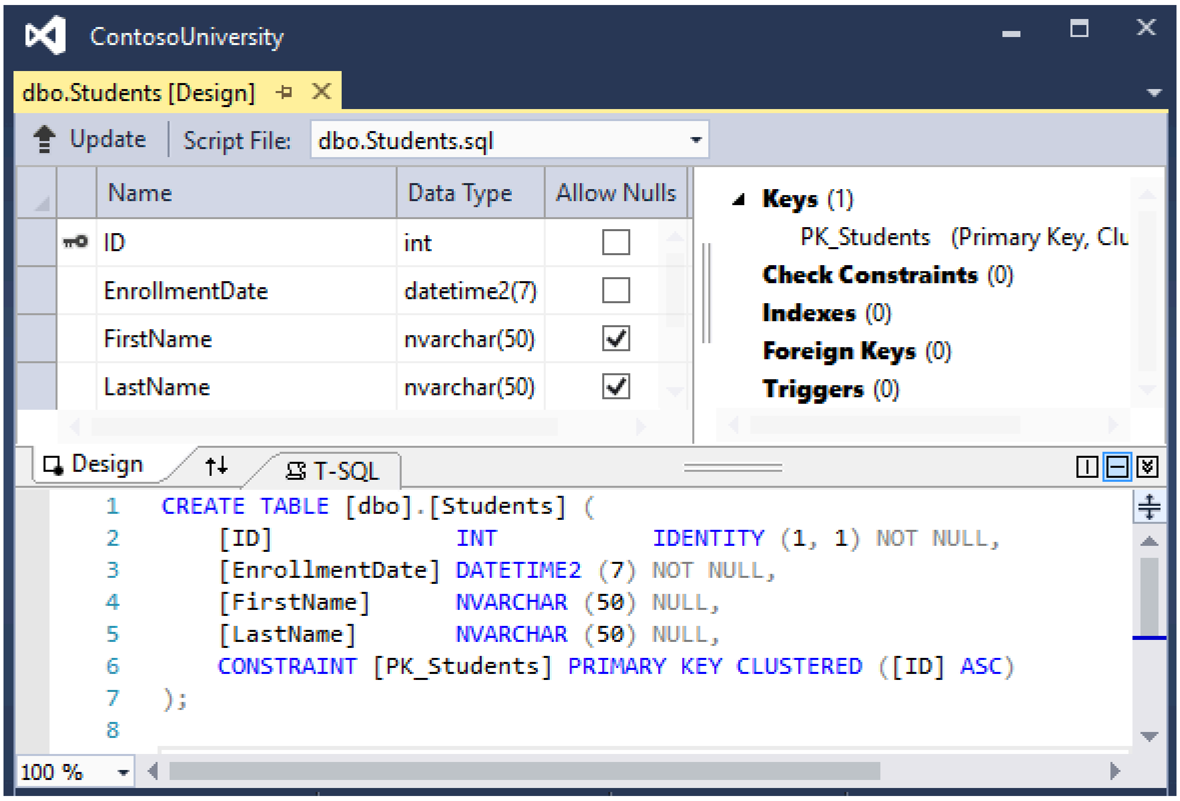
.NET CLI

```
dotnet ef migrations add ColumnFirstName
```

.NET CLI

```
dotnet ef database update
```

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



ContosoUniversity

dbo.Students [Design]

Update | Script File: dbo.Students.sql

Name	Data Type	Allow Nulls
ID	int	☐
EnrollmentDate	datetime2(7)	☐
FirstName	nvarchar(50)	✓
LastName	nvarchar(50)	✓

Keys (1)
PK_Students (Primary Key, Clu
Check Constraints (0)
Indexes (0)
Foreign Keys (0)
Triggers (0)

Design | T-SQL

```
CREATE TABLE [dbo].[Students] (
    [ID]             INT           IDENTITY (1, 1) NOT NULL,
    [EnrollmentDate] DATETIME2 (7) NOT NULL,
    [FirstName]      NVARCHAR (50) NULL,
    [LastName]       NVARCHAR (50) NULL,
    CONSTRAINT [PK_Students] PRIMARY KEY CLUSTERED ([ID] ASC)
);
```

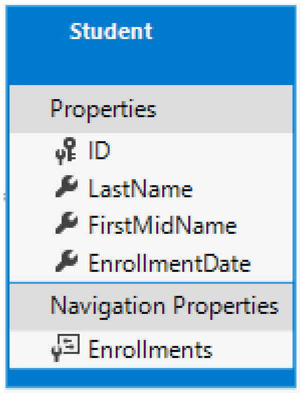
100 %

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

ⓘ Note

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

Changes to Student entity



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

C#

```
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)]
        [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, float, etc.). 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.

```C#
[Display(Name = "Last Name")]
[Required]
```

```
    [StringLength(50, MinimumLength=2)]
    public string LastName { get; set; }
```

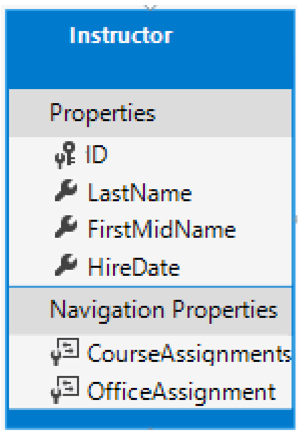
The Display attribute

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

The FullName calculated property

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

Create Instructor entity



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

C#

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

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

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

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

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

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

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

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

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

C#

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

The CourseAssignments and OfficeAssignment navigation properties

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

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

C#

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

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

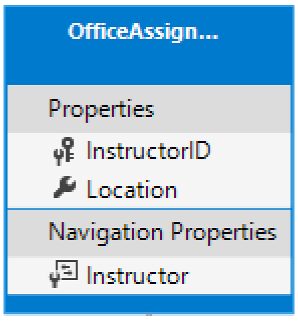
The reason why these are `CourseAssignment` entities is explained below in the section about many-to-many relationships.

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

C#

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

Create OfficeAssignment entity



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

C#

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

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }
        public Instructor Instructor { get; set; }
    }
}
```

The Key attribute

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

C#

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

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

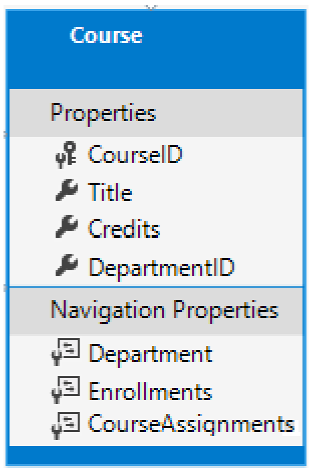
By default, EF treats the key as non-database-generated because the column is for an identifying relationship.

The Instructor navigation property

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

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

Modify Course entity



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

C#

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

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

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

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

        public int DepartmentID { get; set; }

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

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

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

The DatabaseGenerated attribute

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

C#

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

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

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

Foreign key and navigation properties

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

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

C#

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

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

C#

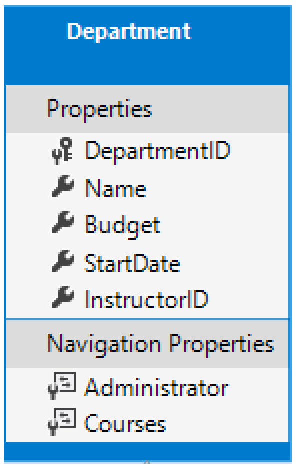
```
public ICollection<Enrollment> Enrollments { get; set; }
```

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

C#

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

Create Department entity



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

C#

```
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

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

C#

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

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

Foreign key and navigation properties

The foreign key and navigation properties reflect the following relationships:

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

C#

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

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

C#

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

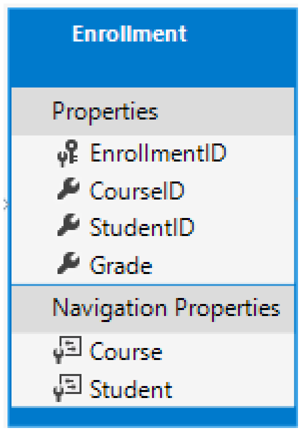
ⓘ Note

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

C#

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

Modify Enrollment entity



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

C#

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

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

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

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

Foreign key and navigation properties

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

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

C#

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

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

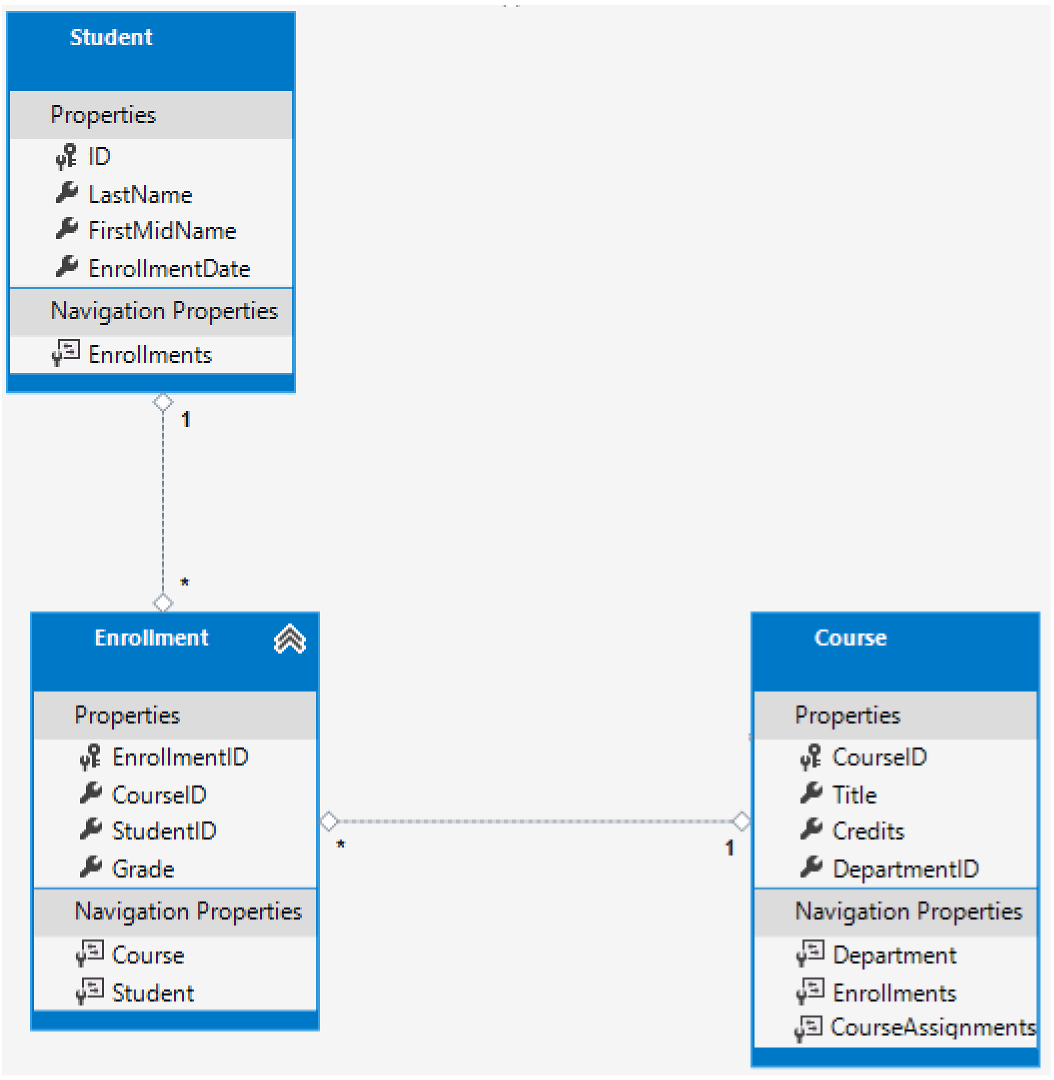
C#

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

Many-to-Many relationships

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

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



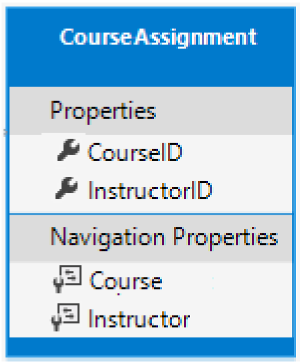
Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

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

EF Core supports implicit join tables for many-to-many relationships, but this tutoral has not been updated to use an implicit join table. See Many-to-Many Relationships, the

Razor Pages version of this tutorial which has been updated.

The CourseAssignment entity



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

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

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

Join entity names

A join table is required in the database for the Instructor-to-Courses many-to-many relationship, and it has to be represented by an entity set. It's common to name a join entity `EntityName1EntityName2`, which in this case would be `CourseInstructor`. However, we recommend that you choose a name that describes the relationship. Data models start out simple and grow, with no-payload joins frequently getting payloads later. If you start with a descriptive entity name, you won't have to change the name later. Ideally, the join entity would have its own natural (possibly single word) name in the business domain. For example, Books and Customers could be linked through Ratings. For this relationship, `CourseAssignment` is a better choice than `CourseInstructor`.

Composite key

Since the foreign keys are not nullable and together uniquely identify each row of the table, there's no need for a separate primary key. The `InstructorID` and `CourseID` properties should function as a composite primary key. The only way to identify composite primary keys to EF is by using the *fluent API* (it can't be done by using attributes). You'll see how to configure the composite primary key in the next section.

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

Update the database context

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

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

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

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>
().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>
().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

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

About a fluent API alternative

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

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

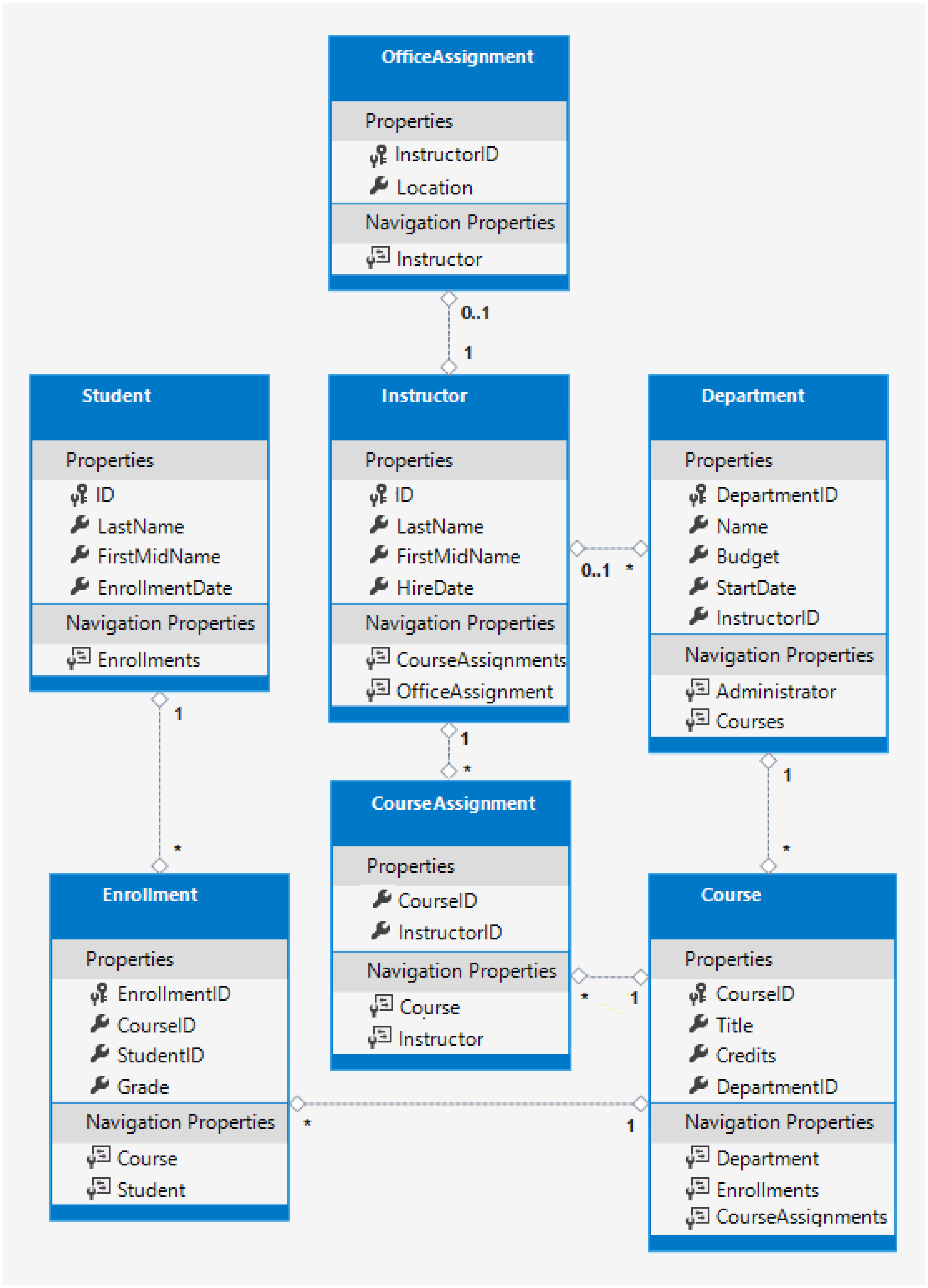
In this tutorial, you're using the fluent API only for database mapping that you can't do with attributes. However, you can also use the fluent API to specify most of the formatting, validation, and mapping rules that you can do by using attributes. Some attributes such as `MinimumLength` can't be applied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

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

For more information about attributes vs. fluent API, see Methods of configuration.

Entity Diagram Showing Relationships

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



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

Seed database with test data

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

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

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

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

            var students = new Student[]
            {
                new Student { FirstMidName = "Carson",   LastName =
"Alexander",
                    EnrollmentDate = DateTime.Parse("2010-09-01") },
                new Student { FirstMidName = "Meredith", LastName =
"Alonso",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Arturo",   LastName = "Anand",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Gytis",    LastName =
"Barzdukas",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Yan",      LastName = "Li",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Peggy",    LastName =
"Justice",
                    EnrollmentDate = DateTime.Parse("2011-09-01") },
                new Student { FirstMidName = "Laura",    LastName =
"Norman",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Nino",     LastName =
"Olivetto",
                    EnrollmentDate = DateTime.Parse("2005-09-01") }
            };

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

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

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

            var departments = new Department[]
            {
                new Department { Name = "English",     Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Abercrombie").ID },
                new Department { Name = "Mathematics", Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Fakhouri").ID },
                new Department { Name = "Engineering", Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Harui").ID },
                new Department { Name = "Economics",   Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Kapoor").ID }
            };
            foreach (Department d in departments)
            {
                context.Departments.Add(d);
            }
            context.SaveChanges();

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

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

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

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

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

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

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

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

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

Add a migration

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

.NET CLI

```
dotnet ef migrations add ComplexDataModel
```

You get a warning about possible data loss.

text

```
An operation was scaffolded that may result in the loss of data. Please
review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'
```

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

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

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

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

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

C#

```csharp
migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);
```

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

C#

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

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

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

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

Save your changes and build the project.

Change the connection string

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

JSON

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

Save your change to `appsettings.json`.

ⓘ Note

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

.NET CLI

```
dotnet ef database drop
```

Update the database

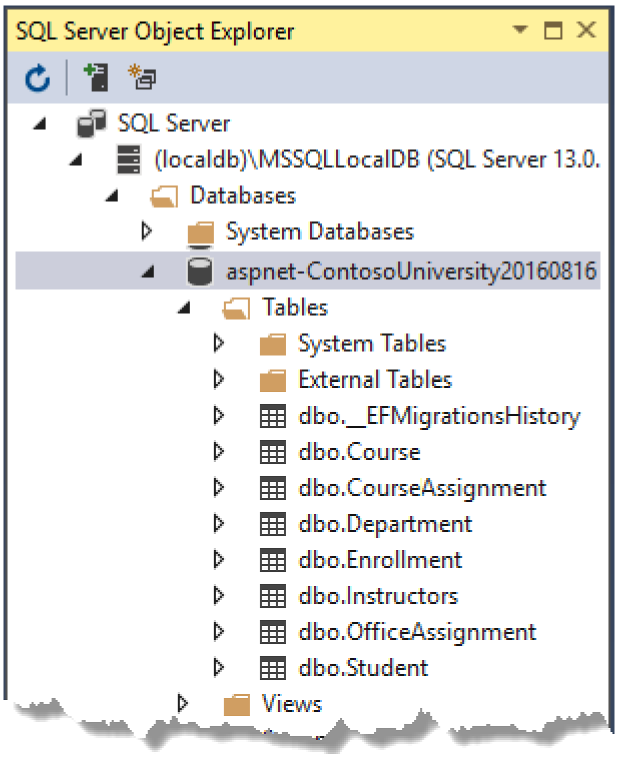
After you have changed the database name or deleted the database, run the `database update` command in the command window to execute the migrations.

.NET CLI

```
dotnet ef database update
```

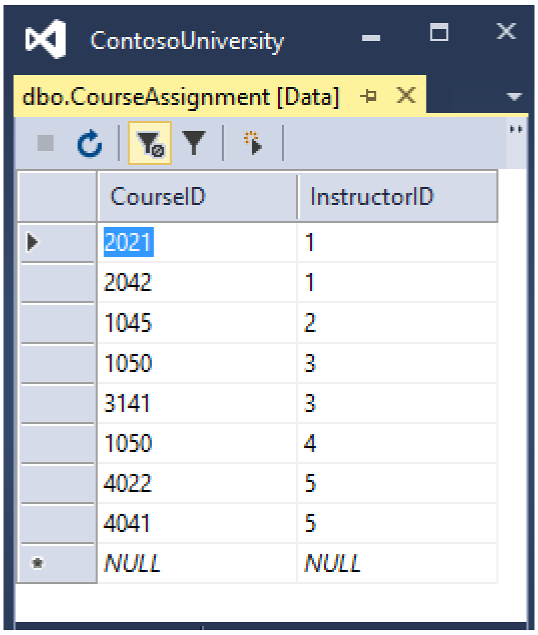
Run the app to cause the `DbInitializer.Initialize` method to run and populate the new database.

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



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

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



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

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Customized the Data model
- ✓ Made changes to Student entity
- ✓ Created Instructor entity
- ✓ Created OfficeAssignment entity
- ✓ Modified Course entity
- ✓ Created Department entity
- ✓ Modified Enrollment entity
- ✓ Updated the database context
- ✓ Seeded database with test data
- ✓ Added a migration
- ✓ Changed the connection string
- ✓ Updated the database

Advance to the next tutorial to learn more about how to access related data.

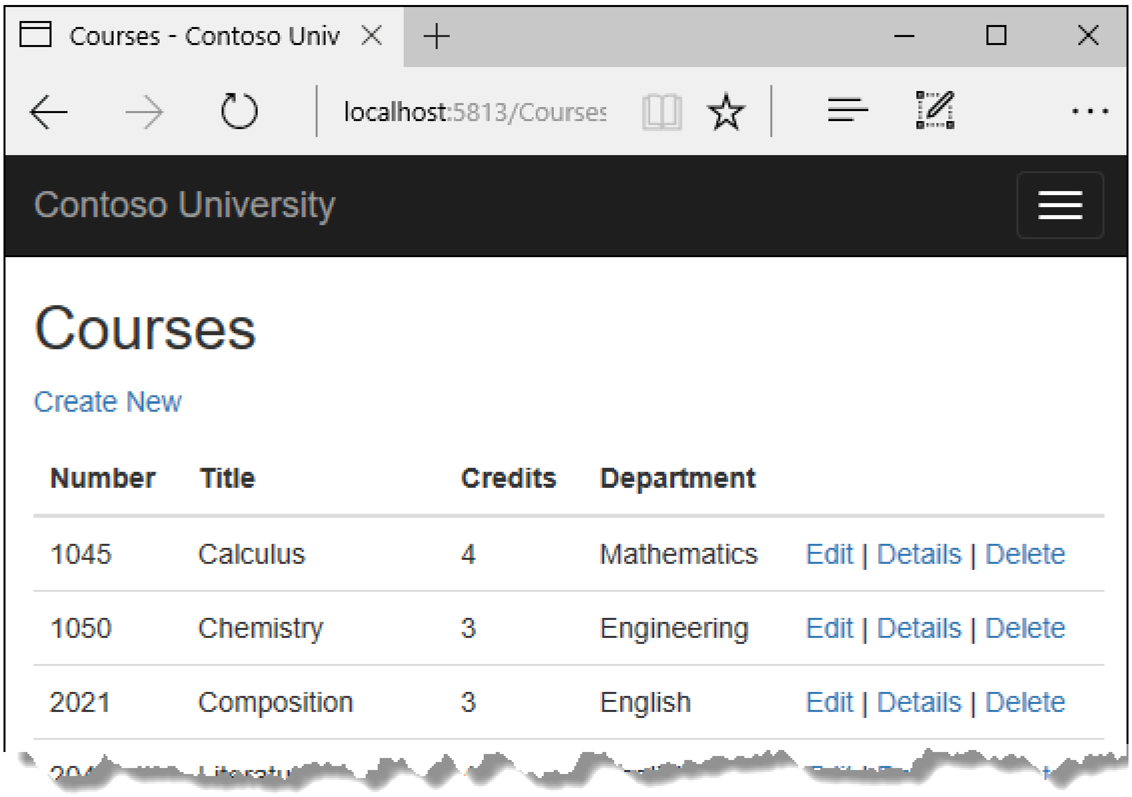
Next: Access related data

Tutorial: Read related data - ASP.NET MVC with EF Core

Article • 04/10/2024

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

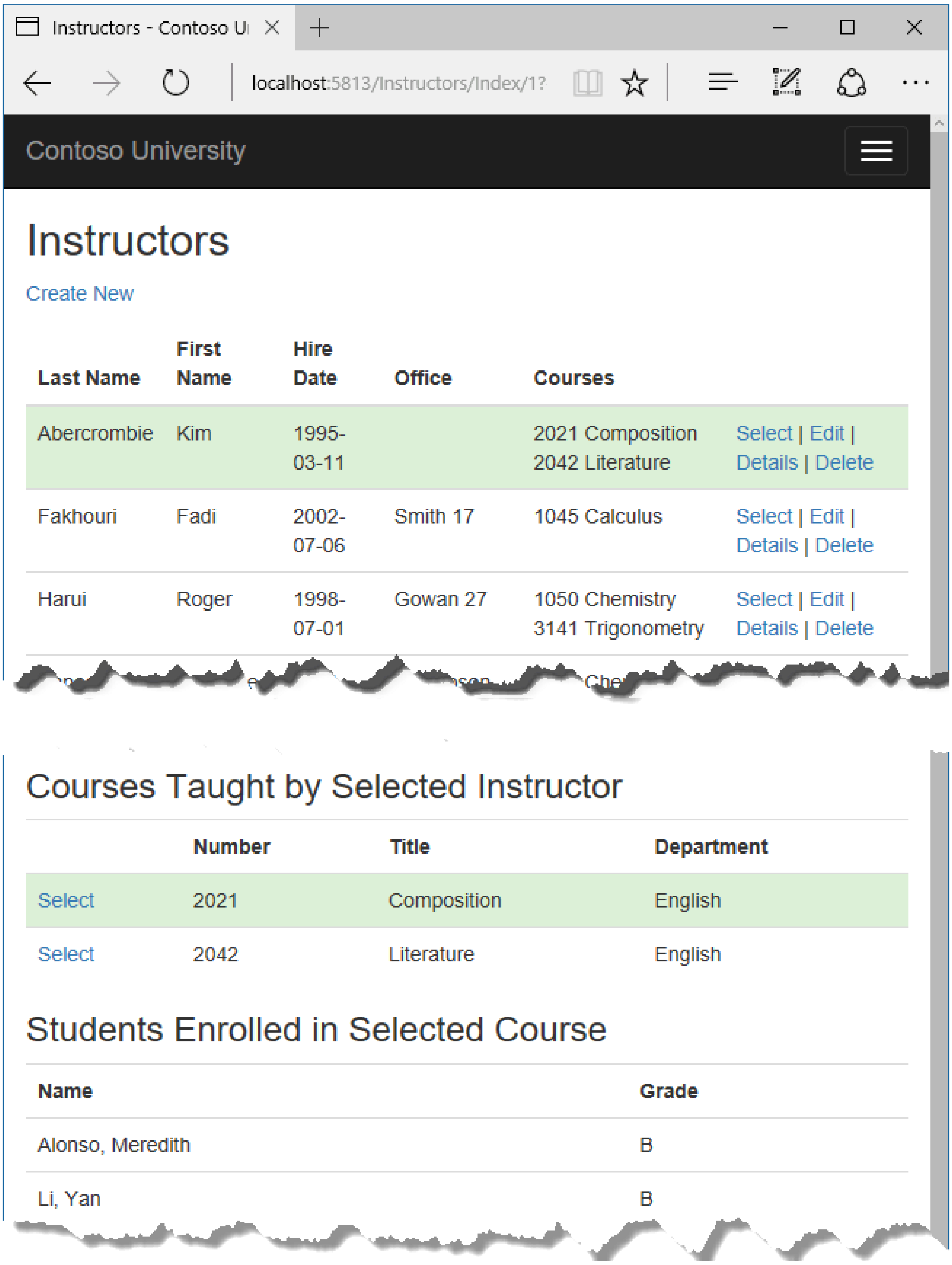
The following illustrations show the pages that you'll work with.



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


Instructors - Contoso U

localhost:5813/Instructors/Index/1?

Contoso University

Instructors

Create New

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

Courses Taught by Selected Instructor

	Number	Title	Department
Select	2021	Composition	English
Select	2042	Literature	English

Students Enrolled in Selected Course

Name	Grade
Alonso, Meredith	B
Li, Yan	B

In this tutorial, you:

- ✓ Learn how to load related data
- ✓ Create a Courses page
- ✓ Create an Instructors page
- ✓ Learn about explicit loading

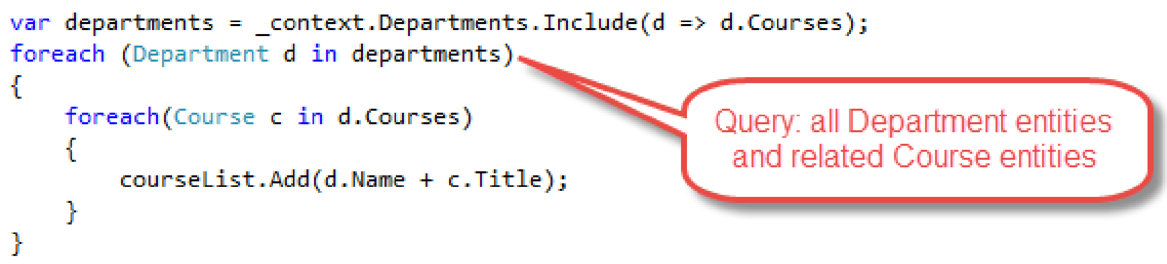
Prerequisites

- Create a complex data model

Learn how to load related data

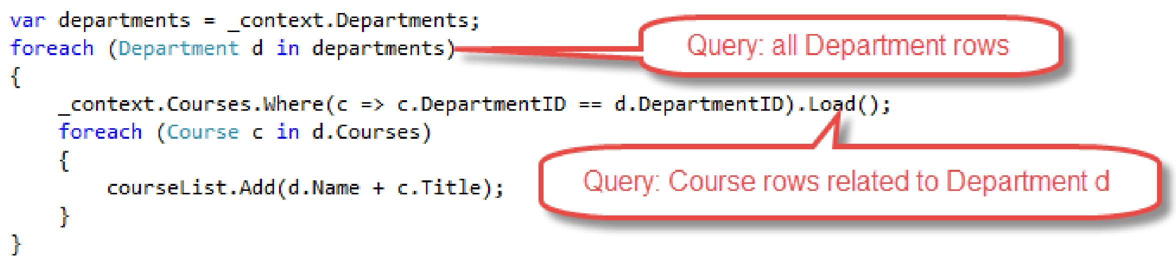
There are several ways that Object-Relational Mapping (ORM) software such as Entity Framework can load related data into the navigation properties of an entity:

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

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

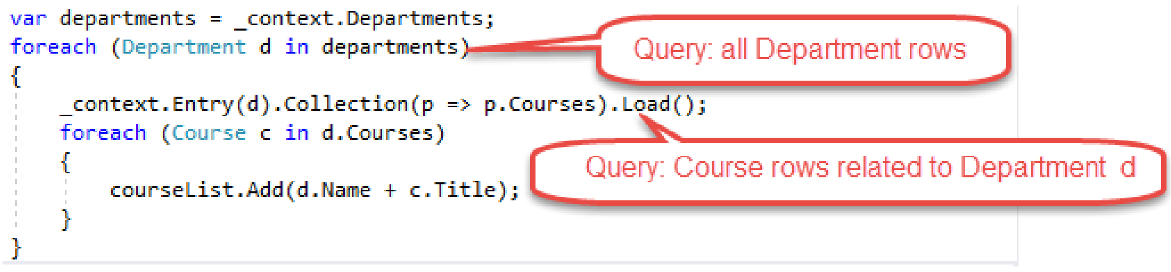
  You can retrieve some of the data in separate queries, and EF "fixes up" the navigation properties. That is, EF automatically adds the separately retrieved entities where they belong in navigation properties of previously retrieved entities. For the query that retrieves related data, you can use the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

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

  Query: all Department rows
  Query: Course rows related to Department d

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



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

Query: all Department rows
Query: Course rows related to Department d

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

Performance considerations

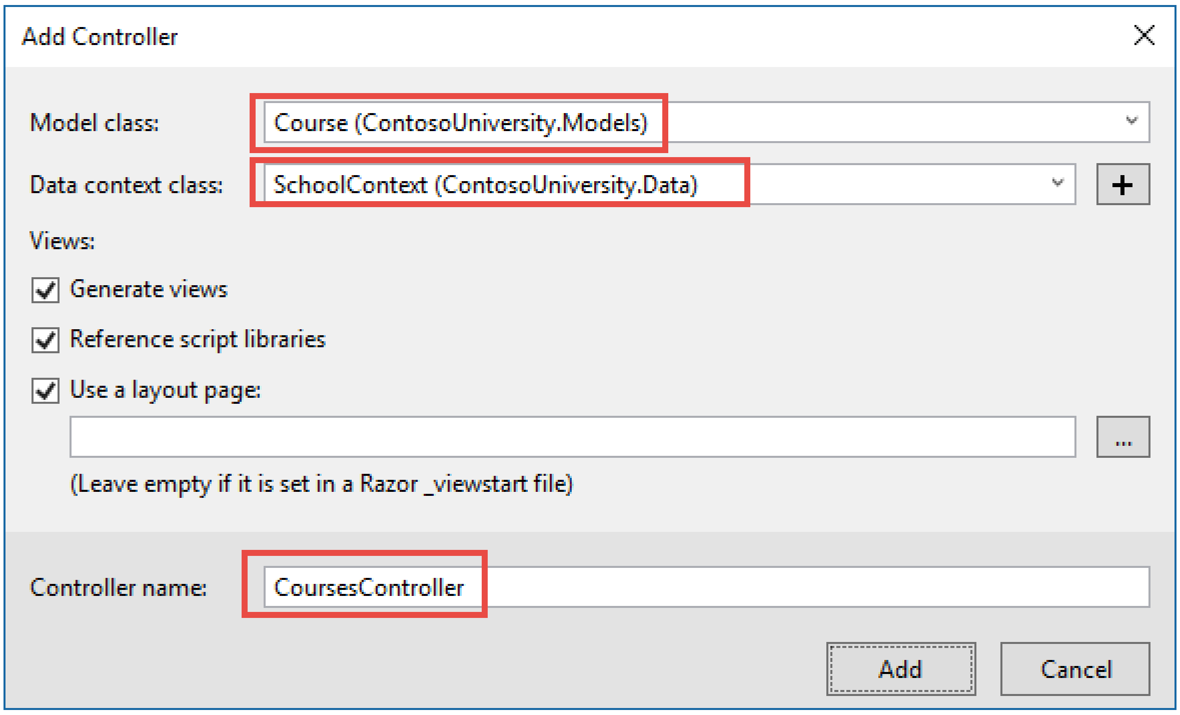
If you know you need related data for every entity retrieved, eager loading often offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. For example, suppose that each department has ten related courses. Eager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

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

Create a Courses page

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

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


Add Controller

Model class: Course (ContosoUniversity.Models)

Data context class: SchoolContext (ContosoUniversity.Data)

Views:

- [✓] Generate views
- [✓] Reference script libraries
- [✓] Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

Controller name: CoursesController

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

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

C#

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

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

CSHTML

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

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

```
<h2>Courses</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.CourseID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Credits)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Department.Name)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-
id="@item.CourseID">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.CourseID">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.CourseID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

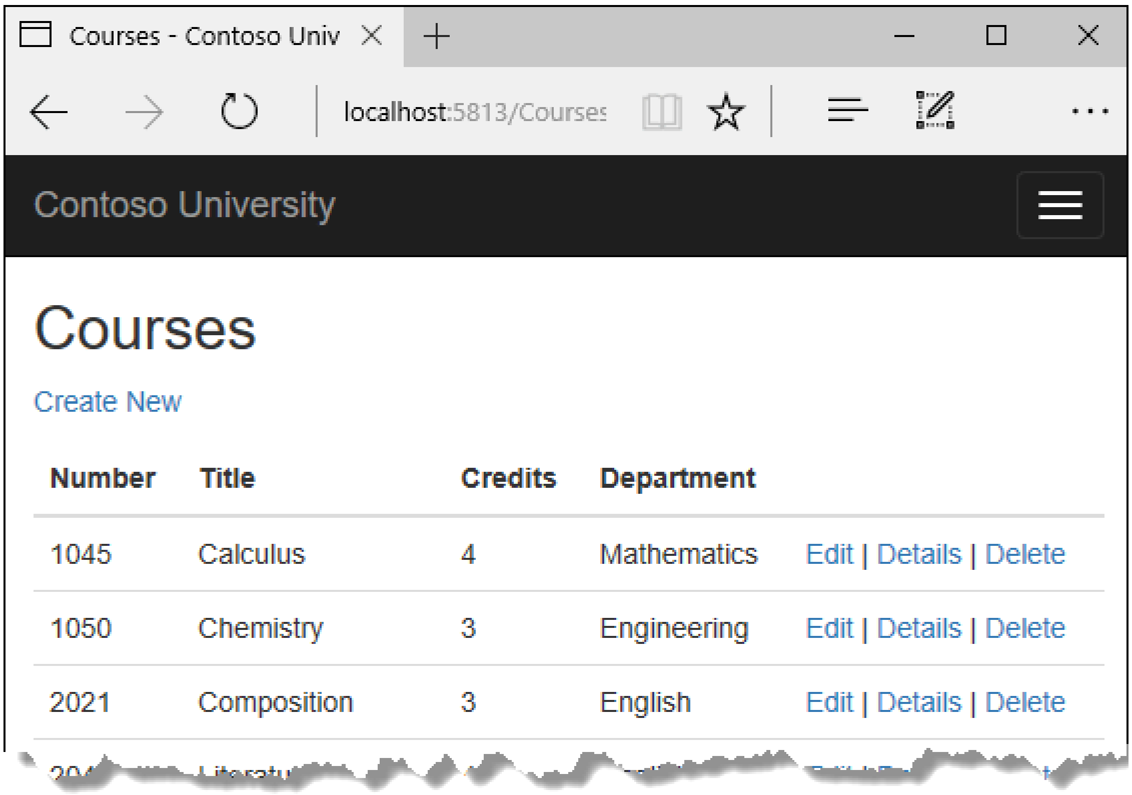
You've made the following changes to the scaffolded code:

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

  HTML

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

Run the app and select the **Courses** tab to see the list with department names.



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

Create an Instructors page

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