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

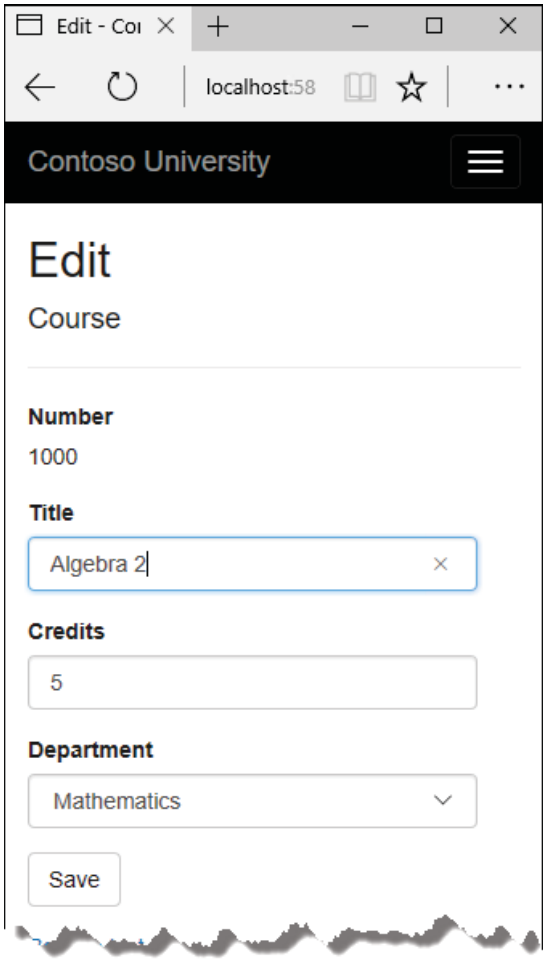
1/29/2018 • 18 min to read • Edit Online

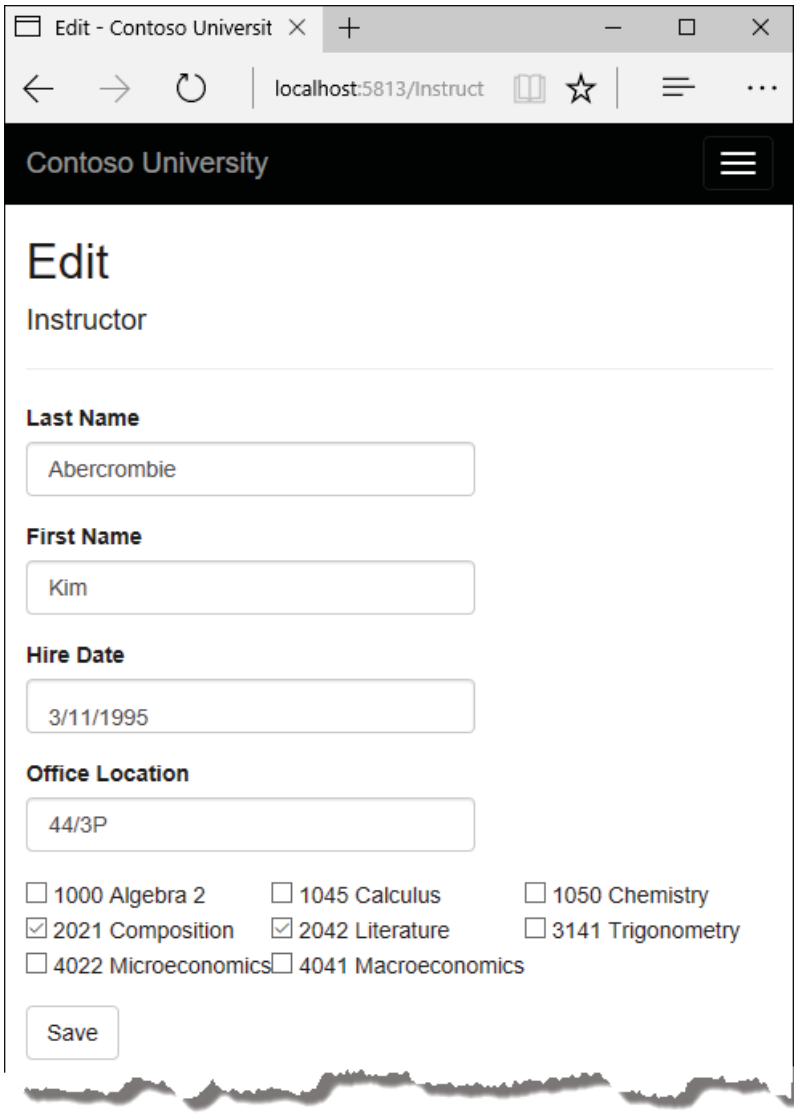
By Tom Dykstra and Rick Anderson

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

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

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





Customize the Create and Edit Pages for Courses

When a new course entity is created, it must have a relationship to an existing department. To facilitate this, the scaffolded code includes controller methods and Create and Edit views that include a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key property, and that's all the Entity Framework needs in order to load the `Department` navigation property with the appropriate Department entity. You'll use the scaffolded code, but change it slightly to add error handling and sort the drop-down list.

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

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

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

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

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

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

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

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

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

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

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

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

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

The HttpGet `Edit` method sets the selected item, based on the ID of the department that's already assigned to the

course being edited:

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

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

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

Add .AsNoTracking to Details and Delete methods

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

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

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

    return View(course);
}
```

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

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

    return View(course);
}
```

Modify the Course views

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

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

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

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

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

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

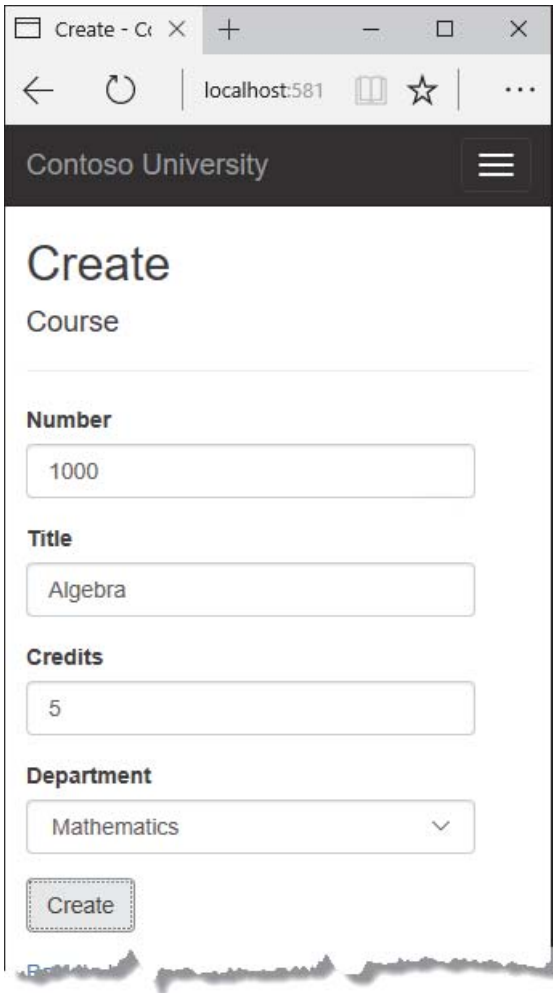
In *Views/Courses/Delete.cshtml*, add a course number field at the top and change department ID to department name.

```
@model ContosoUniversity.Models.Course

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

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.CourseID)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.CourseID)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Credits)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Credits)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
    </dl>

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

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

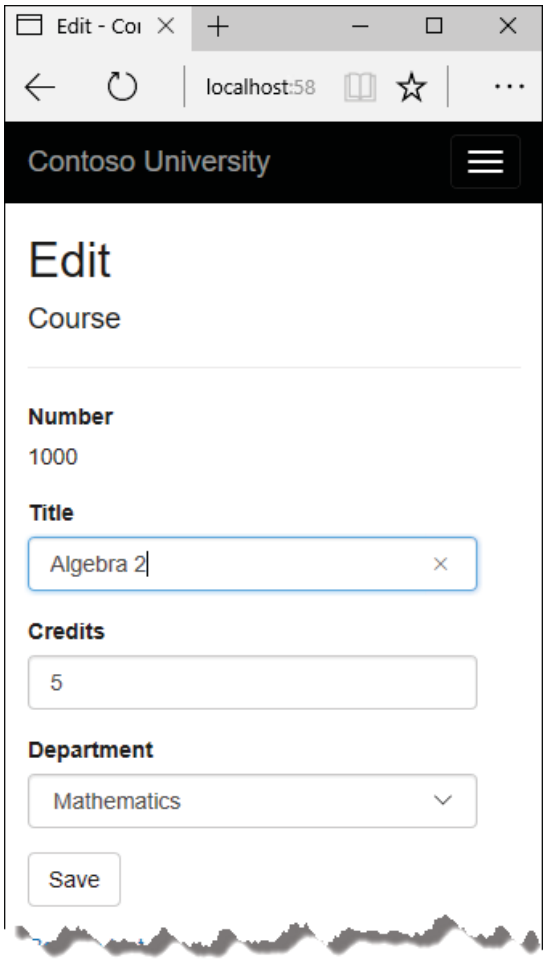
Test the Course pages

Run the app, select the **Courses** tab, click **Create New**, and enter data for a new course:



Click **Create**. The Courses Index page is displayed with the new course added to the list. The department name in the Index page list comes from the navigation property, showing that the relationship was established correctly.

Click **Edit** on a course in the Courses Index page.



Change data on the page and click **Save**. The Courses Index page is displayed with the updated course data.

Add an Edit Page for Instructors

When you edit an instructor record, you want to be able to update the instructor's office assignment. The Instructor entity has a one-to-zero-or-one relationship with the OfficeAssignment entity, which means your code has to handle the following situations:

- If the user clears the office assignment and it originally had a value, delete the OfficeAssignment entity.
- If the user enters an office assignment value and it originally was empty, create a new OfficeAssignment entity.
- If the user changes the value of an office assignment, change the value in an existing OfficeAssignment entity.

Update the Instructors controller

In *InstructorsController.cs*, change the code in the HttpGet `Edit` method so that it loads the Instructor entity's `OfficeAssignment` navigation property and calls `AsNoTracking`:

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

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    return View(instructor);
}
```

Replace the HttpPost `Edit` method with the following code to handle office assignment updates:

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

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .SingleOrDefaultAsync(s => s.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
    {
        if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    return View(instructorToUpdate);
}
```

The code does the following:

- Changes the method name to `EditPost` because the signature is now the same as the HttpGet `Edit` method (the `ActionName` attribute specifies that the `/Edit/` URL is still used).
- Gets the current Instructor entity from the database using eager loading for the `OfficeAssignment`

navigation property. This is the same as what you did in the HttpGet `Edit` method.

- Updates the retrieved Instructor entity with values from the model binder. The `TryUpdateModel` overload ~~enables you to whitelist the properties you want to include. This prevents over-posting, as explained in the~~ second tutorial.

```
if (await TryUpdateModelAsync<Instructor>(
    instructorToUpdate,
    "",
    i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
```

- If the office location is blank, sets the Instructor.OfficeAssignment property to null so that the related row in the OfficeAssignment table will be deleted.

```
if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
{
    instructorToUpdate.OfficeAssignment = null;
}
```

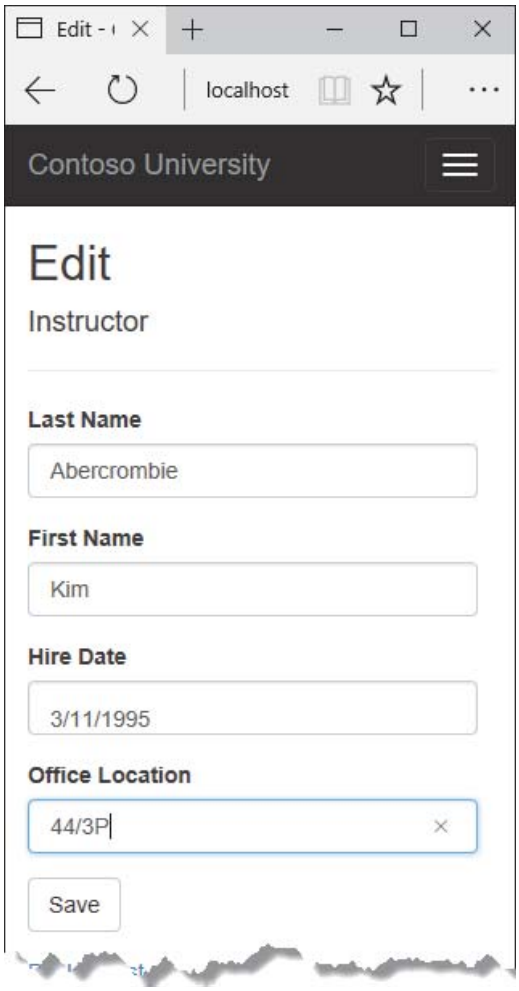
- Saves the changes to the database.

Update the Instructor Edit view

In *Views/Instructors/Edit.cshtml*, add a new field for editing the office location, at the end before the **Save** button:

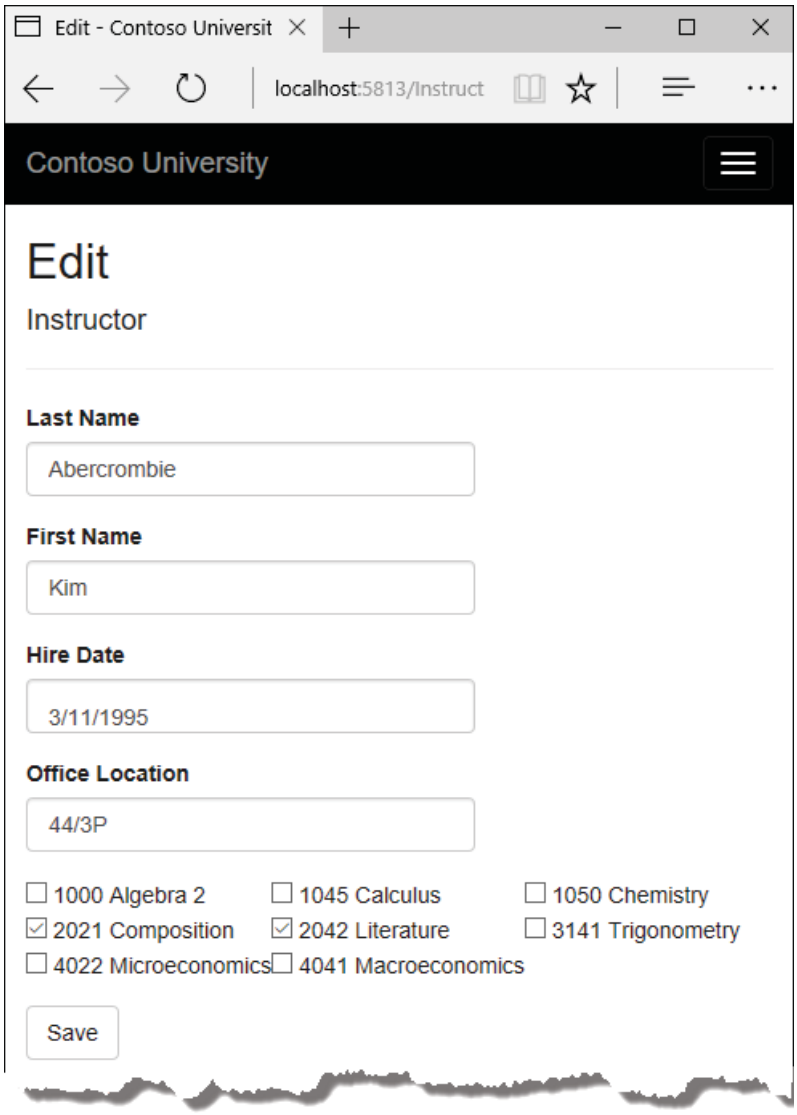
```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label"></label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
</div>
```

Run the app, select the **Instructors** tab, and then click **Edit** on an instructor. Change the **Office Location** and click **Save**.



Add Course assignments to the Instructor Edit page

Instructors may teach any number of courses. Now you'll enhance the Instructor Edit page by adding the ability to change course assignments using a group of check boxes, as shown in the following screen shot:



The relationship between the Course and Instructor entities is many-to-many. To add and remove relationships, you add and remove entities to and from the CourseAssignments join entity set.

The UI that enables you to change which courses an instructor is assigned to is a group of check boxes. A check box for every course in the database is displayed, and the ones that the instructor is currently assigned to are selected. The user can select or clear check boxes to change course assignments. If the number of courses were much greater, you would probably want to use a different method of presenting the data in the view, but you'd use the same method of manipulating a join entity to create or delete relationships.

Update the Instructors controller

To provide data to the view for the list of check boxes, you'll use a view model class.

Create *AssignedCourseData.cs* in the *SchoolViewModels* folder and replace the existing code with the following code:

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

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

In *InstructorsController.cs*, replace the HttpGet `Edit` method with the following code. The changes are highlighted.

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

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

private void PopulateAssignedCourseData(Instructor instructor)
{
    var allCourses = _context.Courses;
    var instructorCourses = new HashSet<int>(instructor.CourseAssignments.Select(c => c.CourseID));
    var viewModel = new List<AssignedCourseData>();
    foreach (var course in allCourses)
    {
        viewModel.Add(new AssignedCourseData
        {
            CourseID = course.CourseID,
            Title = course.Title,
            Assigned = instructorCourses.Contains(course.CourseID)
        });
    }
    ViewData["Courses"] = viewModel;
}
```

The code adds eager loading for the `Courses` navigation property and calls the new `PopulateAssignedCourseData` method to provide information for the check box array using the `AssignedCourseData` view model class.

The code in the `PopulateAssignedCourseData` method reads through all Course entities in order to load a list of courses using the view model class. For each course, the code checks whether the course exists in the instructor's `Courses` navigation property. To create efficient lookup when checking whether a course is assigned to the instructor, the courses assigned to the instructor are put into a `HashSet` collection. The `Assigned` property is set to true for courses the instructor is assigned to. The view will use this property to determine which check boxes must

be displayed as selected. Finally, the list is passed to the view in `ViewData`.

Next, add the code that's executed when the user clicks **Save**. Replace the `EditPost` method with the following code, and add a new method that updates the `Courses` navigation property of the Instructor entity.

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

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
        .SingleOrDefaultAsync(m => m.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
    {
        if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        UpdateInstructorCourses(selectedCourses, instructorToUpdate);
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    UpdateInstructorCourses(selectedCourses, instructorToUpdate);
    PopulateAssignedCourseData(instructorToUpdate);
    return View(instructorToUpdate);
}
```

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

The method signature is now different from the HttpGet `Edit` method, so the method name changes from `EditPost` back to `Edit`.

Since the view doesn't have a collection of Course entities, the model binder can't automatically update the `CourseAssignments` navigation property. Instead of using the model binder to update the `CourseAssignments` navigation property, you do that in the new `UpdateInstructorCourses` method. Therefore you need to exclude the `CourseAssignments` property from model binding. This doesn't require any change to the code that calls `TryUpdateModel` because you're using the whitelisting overload and `CourseAssignments` isn't in the include list.

If no check boxes were selected, the code in `UpdateInstructorCourses` initializes the `CourseAssignments` navigation property with an empty collection and returns:

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the view. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the check box for a course was selected but the course isn't in the `Instructor.CourseAssignments` navigation property, the course is added to the collection in the navigation property.

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

If the check box for a course wasn't selected, but the course is in the `Instructor.CourseAssignments` navigation property, the course is removed from the navigation property.

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

Update the Instructor views

In *Views/Instructors/Edit.cshtml*, add a **Courses** field with an array of check boxes by adding the following code immediately after the `div` elements for the **Office** field and before the `div` element for the **Save** button.

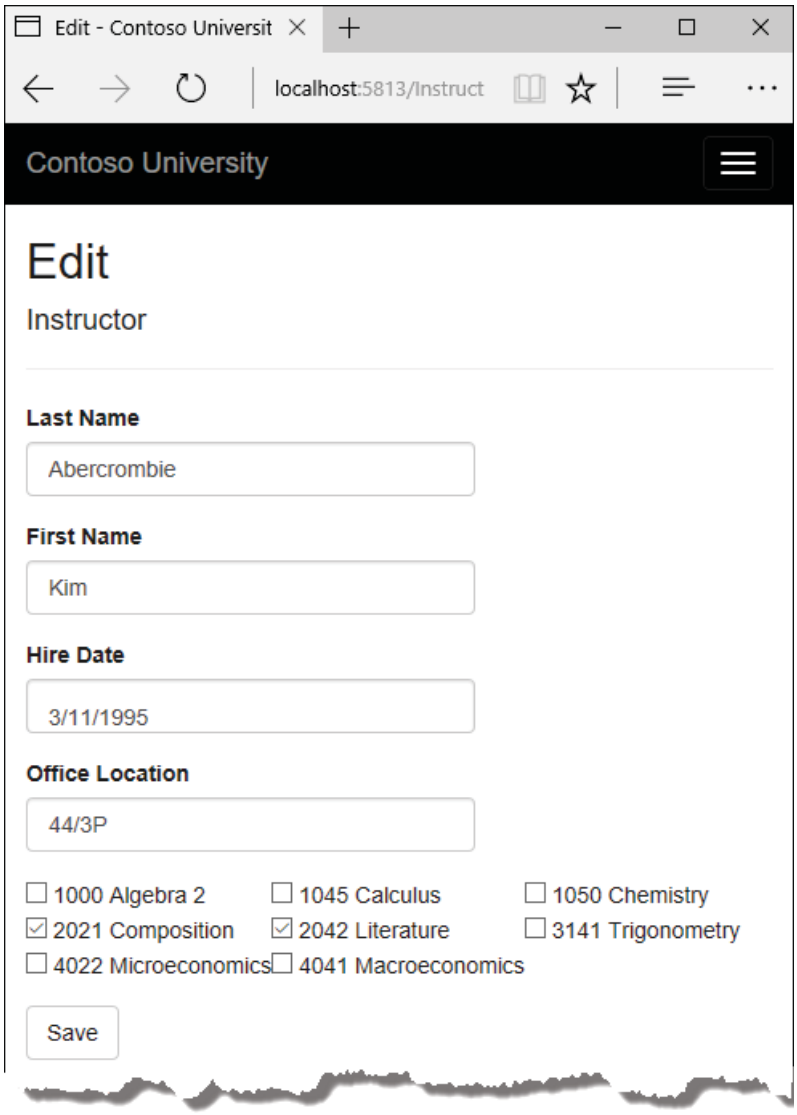
> **NOTE**
>
> When you paste the code in Visual Studio, line breaks will be changed in a way that breaks the code. Press Ctrl+Z one time to undo the automatic formatting. This will fix the line breaks so that they look like what you see here. The indentation doesn't have to be perfect, but the `@</tr><tr>`, `@:<td>`, `@:</td>`, and `@:</tr>` lines must each be on a single line as shown or you'll get a runtime error. With the block of new code selected, press Tab three times to line up the new code with the existing code. You can check the status of this problem here.

```
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;
                    List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                   name="selectedCourses"
                                   value="@course.CourseID"
                                   @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                   @course.CourseID @: @course.Title
                        @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

This code creates an HTML table that has three columns. In each column is a check box followed by a caption that consists of the course number and title. The check boxes all have the same name ("selectedCourses"), which informs the model binder that they're to be treated as a group. The value attribute of each check box is set to the value of `CourseID`. When the page is posted, the model binder passes an array to the controller that consists of the `CourseID` values for only the check boxes which are selected.

When the check boxes are initially rendered, those that are for courses assigned to the instructor have checked attributes, which selects them (displays them checked).

Run the app, select the **Instructors** tab, and click **Edit** on an instructor to see the **Edit** page.



Change some course assignments and click Save. The changes you make are reflected on the Index page.

> **NOTE**
> The approach taken here to edit instructor course data works well when there's a limited number of courses. For collections that are much larger, a different UI and a different updating method would be required.

Update the Delete page

In *InstructorsController.cs*, delete the `DeleteConfirmed` method and insert the following code in its place.

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    Instructor instructor = await _context.Instructors
        .Include(i => i.CourseAssignments)
        .SingleAsync(i => i.ID == id);

    var departments = await _context.Departments
        .Where(d => d.InstructorID == id)
        .ToListAsync();
    departments.ForEach(d => d.InstructorID = null);

    _context.Instructors.Remove(instructor);

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

This code makes the following changes:

- Does eager loading for the `CourseAssignments` navigation property. You have to include this or EF won't know about related `CourseAssignment` entities and won't delete them. To avoid needing to read them here you could configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

Add office location and courses to the Create page

In *InstructorsController.cs*, delete the HttpGet and HttpPost `Create` methods, and then add the following code in their place:

```
public IActionResult Create()
{
    var instructor = new Instructor();
    instructor.CourseAssignments = new List<CourseAssignment>();
    PopulateAssignedCourseData(instructor);
    return View();
}

// POST: Instructors/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("FirstMidName,HireDate,LastName,OfficeAssignment")] Instructor
instructor, string[] selectedCourses)
{
    if (selectedCourses != null)
    {
        instructor.CourseAssignments = new List<CourseAssignment>();
        foreach (var course in selectedCourses)
        {
            var courseToAdd = new CourseAssignment { InstructorID = instructor.ID, CourseID =
int.Parse(course) };
            instructor.CourseAssignments.Add(courseToAdd);
        }
    }
    if (ModelState.IsValid)
    {
        _context.Add(instructor);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}
```

This code is similar to what you saw for the `Edit` methods except that initially no courses are selected. The HttpGet `Create` method calls the `PopulateAssignedCourseData` method not because there might be courses selected but in order to provide an empty collection for the `foreach` loop in the view (otherwise the view code would throw a null reference exception).

The HttpPost `Create` method adds each selected course to the `CourseAssignments` navigation property before it checks for validation errors and adds the new instructor to the database. Courses are added even if there are model errors so that when there are model errors (for an example, the user keyed an invalid date), and the page is redisplayed with an error message, any course selections that were made are automatically restored.

Notice that in order to be able to add courses to the `CourseAssignments` navigation property you have to initialize the property as an empty collection:

```
instructor.CourseAssignments = new List<CourseAssignment>();
```

As an alternative to doing this in controller code, you could do it in the Instructor model by changing the property getter to automatically create the collection if it doesn't exist, as shown in the following example:

```
private ICollection<CourseAssignment> _courseAssignments;
public ICollection<CourseAssignment> CourseAssignments
{
    get
    {
        return _courseAssignments ?? (_courseAssignments = new List<CourseAssignment>());
    }
    set
    {
        _courseAssignments = value;
    }
}
```

If you modify the `CourseAssignments` property in this way, you can remove the explicit property initialization code in the controller.

In *Views/Instructor/Create.cshtml*, add an office location text box and check boxes for courses before the Submit button. As in the case of the Edit page, fix the formatting if Visual Studio reformats the code when you paste it.

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label"></label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;
                    List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                   name="selectedCourses"
                                   value="@course.CourseID"
                                   @(Html.Raw(course.Assigned ? "checked=\"checked\"" : ""))  />
                                   @course.CourseID @:  @course.Title
                            @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

Test by running the app and creating an instructor.

Handling Transactions

As explained in the CRUD tutorial, the Entity Framework implicitly implements transactions. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see Transactions.

Summary

You have now completed the introduction to working with related data. In the next tutorial you'll see how to handle concurrency conflicts.

PREVIOUS NEXT

Handling concurrency conflicts - EF Core with ASP.NET Core MVC tutorial (8 of 10)

1/29/2018 • 18 min to read • Edit Online

By Tom Dykstra and Rick Anderson

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

In earlier tutorials, you learned how to update data. This tutorial shows how to handle conflicts when multiple users update the same entity at the same time.

You'll create web pages that work with the Department entity and handle concurrency errors. The following illustrations show the Edit and Delete pages, including some messages that are displayed if a concurrency conflict occurs.