**ASP.NET Core MVC with EF Core**

<https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/advanced?view=aspnetcore-3.0>

[Tom Dykstra](https://github.com/tdykstra), [Rick Anderson](https://twitter.com/RickAndMSFT)

03/26/2017

## Overview

This tutorial has **not** been updated to ASP.NET Core 3.0. The [Razor Pages version](https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp/intro?view=aspnetcore-3.0) has been updated. For information on when this might be updated, see [this GitHub issue](https://github.com/aspnet/AspNetCore.Docs/issues/13920).

This tutorial teaches ASP.NET Core MVC and Entity Framework Core with controllers and views. [Razor Pages](https://docs.microsoft.com/en-us/aspnet/core/razor-pages/index?view=aspnetcore-3.0) is an alternative programming model that was introduced in ASP.NET Core 2.0. For new development, we recommend Razor Pages over MVC with controllers and views. There is a [Razor Pages](https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp/intro?view=aspnetcore-3.0) 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
* A version available for ASP.NET Core 3.0

## Get Started

This tutorial has **not** been updated to ASP.NET Core 3.0. The [Razor Pages version](https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp/intro?view=aspnetcore-3.0) has been updated. For information on when this might be updated, see [this GitHub issue](https://github.com/aspnet/AspNetCore.Docs/issues/13920).

## Create, Read, Update and Delete

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**](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/advanced?view=aspnetcore-3.0).

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](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro?view=aspnetcore-3.0)

## Customize the Details page

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

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

C#Copy

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

{

if (id == null)

{

return NotFound();

}

var student = await \_context.Students

.Include(s => s.Enrollments)

.ThenInclude(e => e.Course)

.AsNoTracking()

.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](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/read-related-data?view=aspnetcore-3.0) 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#Copy

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{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.

Copy

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

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

Copy

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

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

HTMLCopy

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

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

HTMLCopy

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

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

HTMLCopy

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

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

HTMLCopy

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

For more information about tag helpers, see [Tag Helpers in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-3.0).

### 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:

HTMLCopy

<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:

HTMLCopy

<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>

}

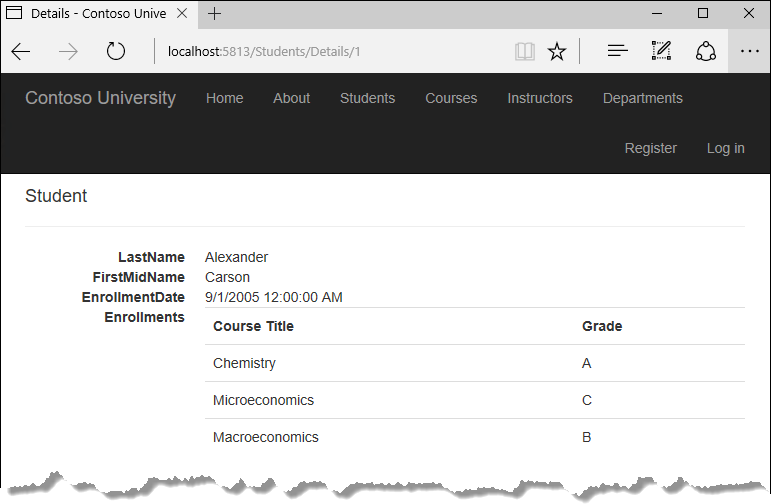
</table>

</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:



## Update the Create page

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

C#Copy

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create(

[Bind("EnrollmentDate,FirstMidName,LastName")] Student student)

{

try

{

if (ModelState.IsValid)

{

\_context.Add(student);

await \_context.SaveChangesAsync();

return RedirectToAction(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)](https://docs.microsoft.com/en-us/aspnet/aspnet/overview/developing-apps-with-windows-azure/building-real-world-cloud-apps-with-windows-azure/monitoring-and-telemetry).

The ValidateAntiForgeryToken attribute helps prevent cross-site request forgery (CSRF) attacks. The token is automatically injected into the view by the [FormTagHelper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-3.0#the-form-tag-helper) and is included when the form is submitted by the user. The token is validated by the ValidateAntiForgeryToken attribute. For more information about CSRF, see [Anti-Request Forgery](https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-3.0).

### Security note about overposting

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

C#Copy

public class Student

{

public int ID { get; set; }

public string LastName { get; set; }

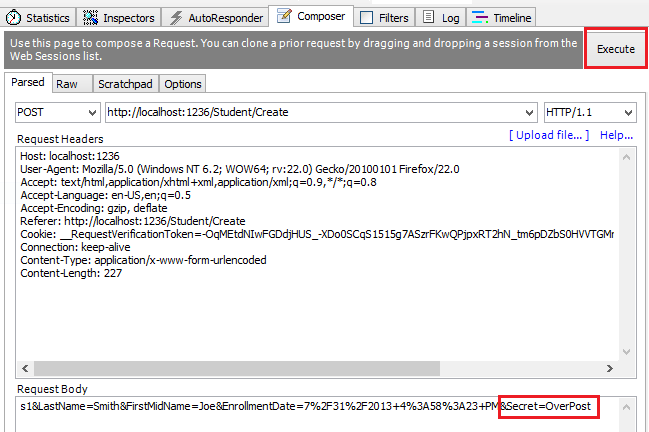
public string FirstMidName { get; set; }

public DateTime EnrollmentDate { get; set; }

public string Secret { get; set; }

}

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



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

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

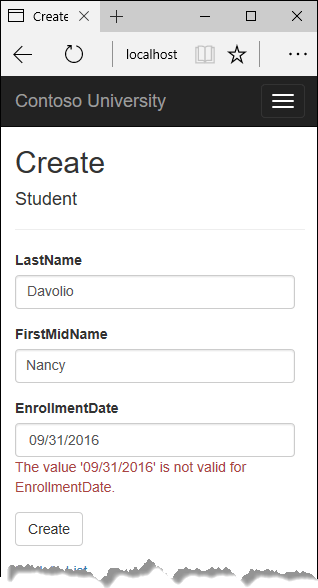
An alternative way to prevent overposting that's preferred by many developers is to use view models rather than entity classes with model binding. Include only the properties you want to update in the view model. Once the MVC model binder has finished, copy the view model properties to the entity instance, 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.



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#Copy

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create(

[Bind("EnrollmentDate,FirstMidName,LastName")] Student student)

{

try

{

if (ModelState.IsValid)

{

\_context.Add(student);

await \_context.SaveChangesAsync();

return RedirectToAction(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 SingleOrDefaultAsync method to retrieve the selected Student entity, as you saw in the Details method. You don't need to change this method.

### Recommended HttpPost Edit code: Read and update

Replace the HttpPost Edit action method with the following code.

C#Copy

[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](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-3.0). The Entity Framework's automatic change tracking sets the Modified flag on the fields that are changed by form input. When the SaveChanges method is called, the Entity Framework creates SQL statements to update the database row. Concurrency conflicts are ignored, and only the table columns that were updated by the user are updated in the database. (A later tutorial shows how to handle concurrency conflicts.)

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

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

### Alternative HttpPost Edit code: Create and attach

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

C#Copy

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

{

if (id != student.ID)

{

return NotFound();

}

if (ModelState.IsValid)

{

try

{

\_context.Update(student);

await \_context.SaveChangesAsync();

return RedirectToAction(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.

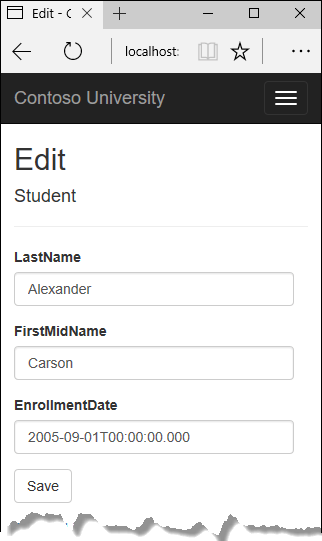
In a web app, the DbContext that initially reads an entity and displays its data to be edited is disposed after a page is rendered. When the HttpPost Edit action method is called, a new web request is made and you have a new instance of the DbContext. If you re-read the entity in that new context, you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



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

## Update the Delete page

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

As you saw for update and create operations, delete operations require two action methods. The method that'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#Copy

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#Copy

[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#Copy

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeleteConfirmed(int id)

{

try

{

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

\_context.Entry(studentToDelete).State = EntityState.Deleted;

await \_context.SaveChangesAsync();

return RedirectToAction(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:

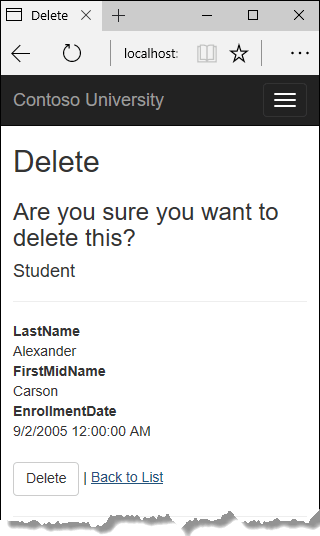
HTMLCopy

<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:



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](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.0) takes care of that task for you.

In Startup.cs, you call the [AddDbContext extension method](https://github.com/aspnet/EntityFrameworkCore/blob/03bcb5122e3f577a84498545fcf130ba79a3d987/src/Microsoft.EntityFrameworkCore/EntityFrameworkServiceCollectionExtensions.cs) 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](https://docs.microsoft.com/en-us/ef/core/saving/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](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/read-related-data?view=aspnetcore-3.0). 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](https://docs.microsoft.com/en-us/ef/core/querying/tracking).

## Next steps

In this tutorial, you:

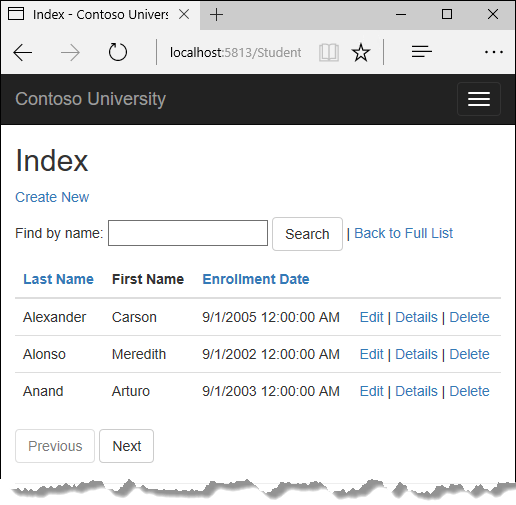
* Customized the Details page
* Updated the Create page
* Updated the Edit page
* Updated the Delete page
* Closed database connections

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

## Sort, filter, page and group

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.



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](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/crud?view=aspnetcore-3.0)

## 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:

C#Copy

public async Task<IActionResult> Index(string sortOrder)

{

ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name\_desc" : "";

ViewData["DateSortParm"] = sortOrder == "Date" ? "date\_desc" : "Date";

var students = from s in \_context.Students

select s;

switch (sortOrder)

{

case "name\_desc":

students = students.OrderByDescending(s => s.LastName);

break;

case "Date":

students = students.OrderBy(s => s.EnrollmentDate);

break;

case "date\_desc":

students = students.OrderByDescending(s => s.EnrollmentDate);

break;

default:

students = students.OrderBy(s => s.LastName);

break;

}

return View(await students.AsNoTracking().ToListAsync());

}

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

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

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

C#Copy

public async Task<IActionResult> Index(string sortOrder)

{

ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name\_desc" : "";

ViewData["DateSortParm"] = sortOrder == "Date" ? "date\_desc" : "Date";

var students = from s in \_context.Students

select s;

switch (sortOrder)

{

case "name\_desc":

students = students.OrderByDescending(s => s.LastName);

break;

case "Date":

students = students.OrderBy(s => s.EnrollmentDate);

break;

case "date\_desc":

students = students.OrderByDescending(s => s.EnrollmentDate);

break;

default:

students = students.OrderBy(s => s.LastName);

break;

}

return View(await students.AsNoTracking().ToListAsync());

}

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

| **Current sort order** | **Last Name Hyperlink** | **Date Hyperlink** |
| --- | --- | --- |
| Last Name ascending | descending | ascending |
| Last Name descending | ascending | ascending |
| Date ascending | ascending | descending |
| Date descending | ascending | ascending |

The method uses LINQ to Entities to specify the column to sort by. The code creates an IQueryable variable before the switch statement, modifies it in the switch statement, and calls the ToListAsync method after the switch statement. When you create and modify IQueryable variables, no query is sent to the database. The query 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](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/advanced?view=aspnetcore-3.0#dynamic-linq) shows how to write code that lets you pass the name of the OrderBy column in a string variable.

### Add column heading hyperlinks to the Student Index view

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

HTMLCopy

@model IEnumerable<ContosoUniversity.Models.Student>

@{

ViewData["Title"] = "Index";

}

<h2>Index</h2>

<p>

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

</p>

<table class="table">

<thead>

<tr>

<th>

<a asp-action="Index" asp-route-sortOrder="@ViewData["NameSortParm"]">@Html.DisplayNameFor(model => model.LastName)</a>

</th>

<th>

@Html.DisplayNameFor(model => model.FirstMidName)

</th>

<th>

<a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]">@Html.DisplayNameFor(model => model.EnrollmentDate)</a>

</th>

<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>

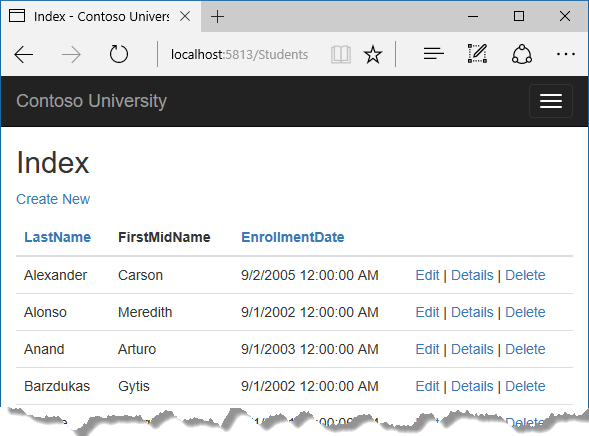
}

</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.



## 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#Copy

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

{

ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name\_desc" : "";

ViewData["DateSortParm"] = sortOrder == "Date" ? "date\_desc" : "Date";

ViewData["CurrentFilter"] = searchString;

var students = from s in \_context.Students

select s;

if (!String.IsNullOrEmpty(searchString))

{

students = students.Where(s => s.LastName.Contains(searchString)

|| s.FirstMidName.Contains(searchString));

}

switch (sortOrder)

{

case "name\_desc":

students = students.OrderByDescending(s => s.LastName);

break;

case "Date":

students = students.OrderBy(s => s.EnrollmentDate);

break;

case "date\_desc":

students = students.OrderByDescending(s => s.EnrollmentDate);

break;

default:

students = students.OrderBy(s => s.LastName);

break;

}

return View(await students.AsNoTracking().ToListAsync());

}

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

**Note**

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

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

HTMLCopy

<p>

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

</p>

<form asp-action="Index" method="get">

<div class="form-actions no-color">

<p>

Find by name: <input type="text" name="SearchString" value="@ViewData["currentFilter"]" />

<input type="submit" value="Search" class="btn btn-default" /> |

<a asp-action="Index">Back to Full List</a>

</p>

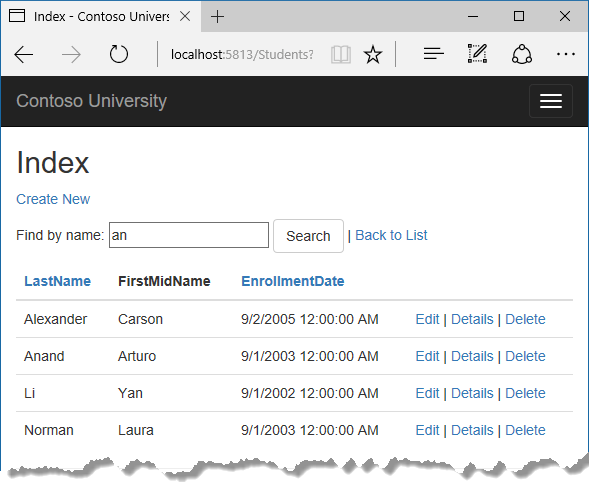
</div>

</form>

<table class="table">

This code uses the <form> [tag helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-3.0) 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.



Notice that the URL contains the search string.

HTMLCopy

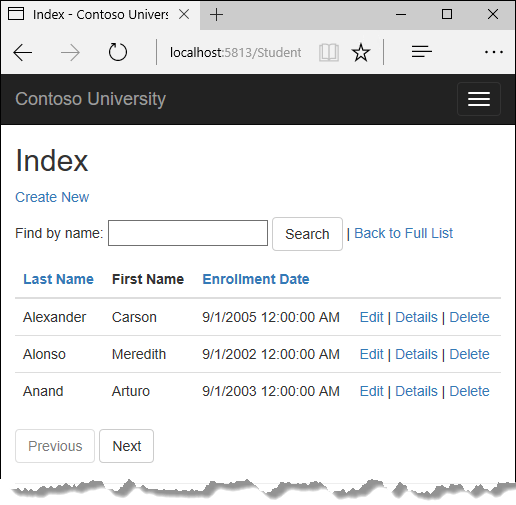
http://localhost:5813/Students?SearchString=an

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

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

## Add paging 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.



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

C#Copy

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

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

{

get

{

return (PageIndex > 1);

}

}

public bool HasNextPage

{

get

{

return (PageIndex < TotalPages);

}

}

public static async Task<PaginatedList<T>> CreateAsync(IQueryable<T> source, int pageIndex, int pageSize)

{

var count = await source.CountAsync();

var items = await source.Skip((pageIndex - 1) \* pageSize).Take(pageSize).ToListAsync();

return new PaginatedList<T>(items, count, pageIndex, pageSize);

}

}

}

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

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

## Add paging to Index method

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

C#Copy

public async Task<IActionResult> Index(

string sortOrder,

string currentFilter,

string searchString,

int? 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#Copy

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#Copy

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#Copy

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.

HTMLCopy

@model PaginatedList<ContosoUniversity.Models.Student>

@{

ViewData["Title"] = "Index";

}

<h2>Index</h2>

<p>

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

</p>

<form asp-action="Index" method="get">

<div class="form-actions no-color">

<p>

Find by name: <input type="text" name="SearchString" value="@ViewData["CurrentFilter"]" />

<input type="submit" value="Search" class="btn btn-default" /> |

<a asp-action="Index">Back to Full List</a>

</p>

</div>

</form>

<table class="table">

<thead>

<tr>

<th>

<a asp-action="Index" asp-route-sortOrder="@ViewData["NameSortParm"]" asp-route-currentFilter="@ViewData["CurrentFilter"]">Last Name</a>

</th>

<th>

First Name

</th>

<th>

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

</th>

<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:

HTMLCopy

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

The paging buttons are displayed by tag helpers:

HTMLCopy

<a asp-action="Index"

asp-route-sortOrder="@ViewData["CurrentSort"]"

asp-route-pageNumber="@(Model.PageIndex - 1)"

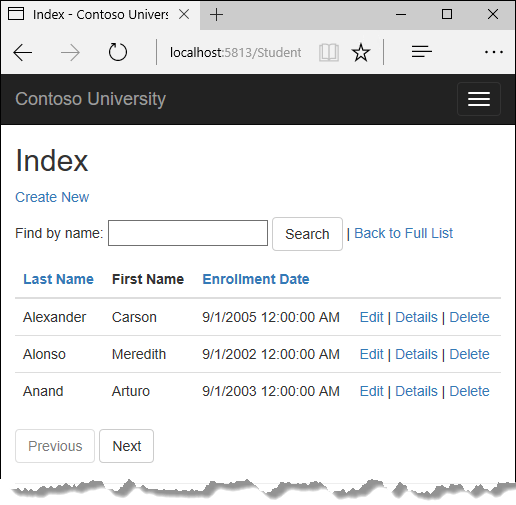
asp-route-currentFilter="@ViewData["CurrentFilter"]"

class="btn btn-default @prevDisabled">

Previous

</a>

Run the app and go to the Students page.



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#Copy

using System;

using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels

{

public class EnrollmentDateGroup

{

[DataType(DataType.Date)]

public DateTime? EnrollmentDate { get; set; }

public int StudentCount { get; set; }

}

}

### Modify the Home Controller

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

C#Copy

using Microsoft.EntityFrameworkCore;

using ContosoUniversity.Data;

using ContosoUniversity.Models.SchoolViewModels;

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

C#Copy

public class HomeController : Controller

{

private readonly SchoolContext \_context;

public HomeController(SchoolContext context)

{

\_context = context;

}

Add an About method with the following code:

C#Copy

public async Task<ActionResult> About()

{

IQueryable<EnrollmentDateGroup> data =

from student in \_context.Students

group student by student.EnrollmentDate into dateGroup

select new EnrollmentDateGroup()

{

EnrollmentDate = dateGroup.Key,

StudentCount = dateGroup.Count()

};

return View(await data.AsNoTracking().ToListAsync());

}

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

### Create the About View

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

HTMLCopy

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

@{

ViewData["Title"] = "Student Body Statistics";

}

<h2>Student Body Statistics</h2>

<table>

<tr>

<th>

Enrollment Date

</th>

<th>

Students

</th>

</tr>

@foreach (var item in Model)

{

<tr>

<td>

@Html.DisplayFor(modelItem => item.EnrollmentDate)

</td>

<td>

@item.StudentCount

</td>

</tr>

}

</table>

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

## 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.

## Migrations

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
* Change the connection string
* Create an initial migration
* Examine Up and Down methods
* Learn about the data model snapshot
* Apply the migration

## Prerequisites

* [Sorting, filtering, and paging](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/sort-filter-page?view=aspnetcore-3.0)

## 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 command-line interface (CLI). These tutorials show how to use CLI commands. Information about the PMC is at [the end of this tutorial](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/migrations?view=aspnetcore-3.0#pmc).

## Change the connection string

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

JSONCopy

{

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity2;Trusted\_Connection=True;MultipleActiveResultSets=true"

},

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

**Note**

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

.NET Core CLICopy

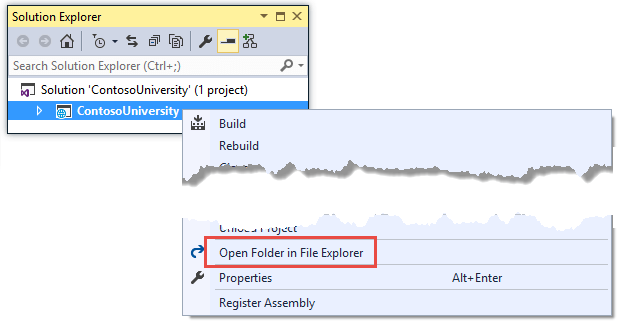
dotnet ef database drop

The following section explains how to run CLI commands.

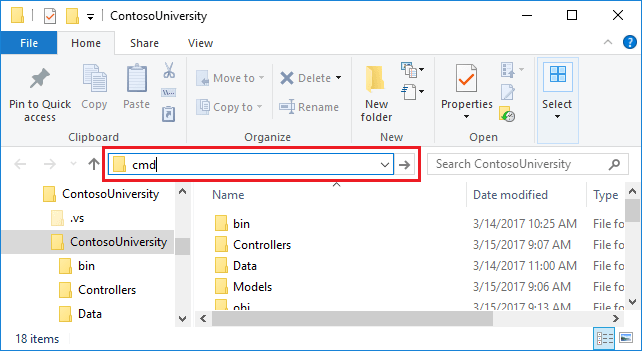
## Create an initial migration

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

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



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



Enter the following command in the command window:

.NET Core CLICopy

dotnet ef migrations add InitialCreate

You see output like the following in the command window:

consoleCopy

info: Microsoft.EntityFrameworkCore.Infrastructure[10403]

Entity Framework Core 2.2.0-rtm-35687 initialized 'SchoolContext' using provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: None

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

**Note**

If you see an error message No executable found matching command "dotnet-ef", see [**this blog post**](https://thedatafarm.com/data-access/no-executable-found-matching-command-dotnet-ef/) for help troubleshooting.

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

## Examine Up and Down methods

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

C#Copy

public partial class InitialCreate : Migration

{

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.CreateTable(

name: "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 changed the name of the database in the connection string 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.

When deleting a migration, use the [dotnet ef migrations remove](https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/dotnet#dotnet-ef-migrations-remove) command. dotnet ef migrations remove deletes the migration and ensures the snapshot is correctly reset.

See [EF Core Migrations in Team Environments](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/teams) 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 Core CLICopy

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](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/index?view=aspnetcore-3.0).

textCopy

info: Microsoft.EntityFrameworkCore.Infrastructure[10403]

Entity Framework Core 2.2.0-rtm-35687 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']

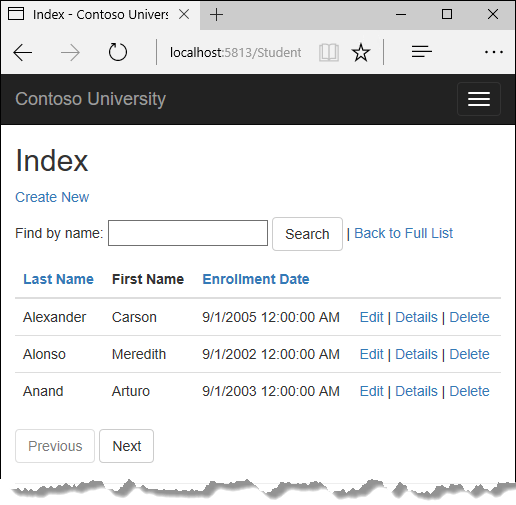
INSERT INTO [\_\_EFMigrationsHistory] ([MigrationId], [ProductVersion])

VALUES (N'20190327172701\_InitialCreate', N'2.2.0-rtm-35687');

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.



## Compare CLI and PMC

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

The EF commands for the PMC commands are in the [Microsoft.EntityFrameworkCore.Tools](https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Tools) package. This package is included in the [Microsoft.AspNetCore.App metapackage](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/metapackage-app?view=aspnetcore-3.0), 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 Core CLI](https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/dotnet).

For more information about the PMC commands, see [Package Manager Console (Visual Studio)](https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/powershell).

## Next step

In this tutorial, you:

* Learned about migrations
* Learned about NuGet migration packages
* Changed the connection string
* Created an initial migration
* Examined Up and Down methods
* Learned about the data model snapshot
* Applied the migration

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 a complex data model

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
* Change the connection string
* Create an initial migration
* Examine Up and Down methods
* Learn about the data model snapshot
* Apply the migration

## Prerequisites

* [Sorting, filtering, and paging](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/sort-filter-page?view=aspnetcore-3.0)

## 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 command-line interface (CLI). These tutorials show how to use CLI commands. Information about the PMC is at [the end of this tutorial](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/migrations?view=aspnetcore-3.0#pmc).

## Change the connection string

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

JSONCopy

{

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity2;Trusted\_Connection=True;MultipleActiveResultSets=true"

},

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

**Note**

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

.NET Core CLICopy

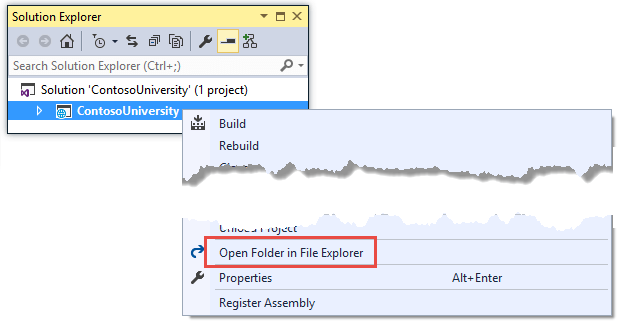
dotnet ef database drop

The following section explains how to run CLI commands.

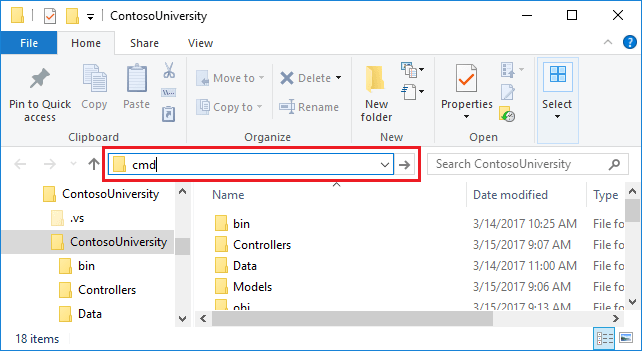
## Create an initial migration

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

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



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



Enter the following command in the command window:

.NET Core CLICopy

dotnet ef migrations add InitialCreate

You see output like the following in the command window:

consoleCopy

info: Microsoft.EntityFrameworkCore.Infrastructure[10403]

Entity Framework Core 2.2.0-rtm-35687 initialized 'SchoolContext' using provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: None

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

**Note**

If you see an error message No executable found matching command "dotnet-ef", see [**this blog post**](https://thedatafarm.com/data-access/no-executable-found-matching-command-dotnet-ef/) for help troubleshooting.

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

## Examine Up and Down methods

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

C#Copy

public partial class InitialCreate : Migration

{

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.CreateTable(

name: "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 changed the name of the database in the connection string 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.

When deleting a migration, use the [dotnet ef migrations remove](https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/dotnet#dotnet-ef-migrations-remove) command. dotnet ef migrations remove deletes the migration and ensures the snapshot is correctly reset.

See [EF Core Migrations in Team Environments](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/teams) 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 Core CLICopy

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](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/index?view=aspnetcore-3.0).

textCopy

info: Microsoft.EntityFrameworkCore.Infrastructure[10403]

Entity Framework Core 2.2.0-rtm-35687 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']

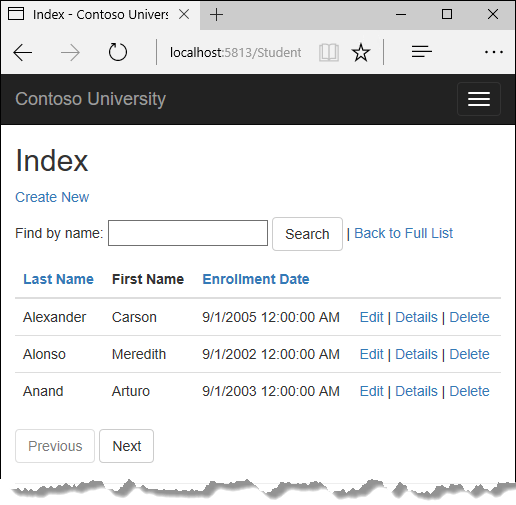
INSERT INTO [\_\_EFMigrationsHistory] ([MigrationId], [ProductVersion])

VALUES (N'20190327172701\_InitialCreate', N'2.2.0-rtm-35687');

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.



## Compare CLI and PMC

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

The EF commands for the PMC commands are in the [Microsoft.EntityFrameworkCore.Tools](https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Tools) package. This package is included in the [Microsoft.AspNetCore.App metapackage](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/metapackage-app?view=aspnetcore-3.0), 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 Core CLI](https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/dotnet).

For more information about the PMC commands, see [Package Manager Console (Visual Studio)](https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/powershell).

## Next step

In this tutorial, you:

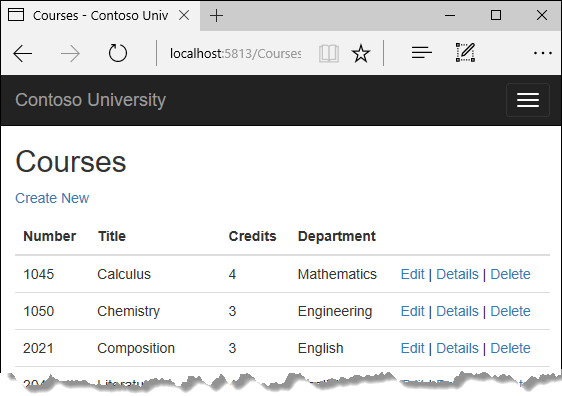
* Learned about migrations
* Learned about NuGet migration packages
* Changed the connection string
* Created an initial migration
* Examined Up and Down methods
* Learned about the data model snapshot
* Applied the migration

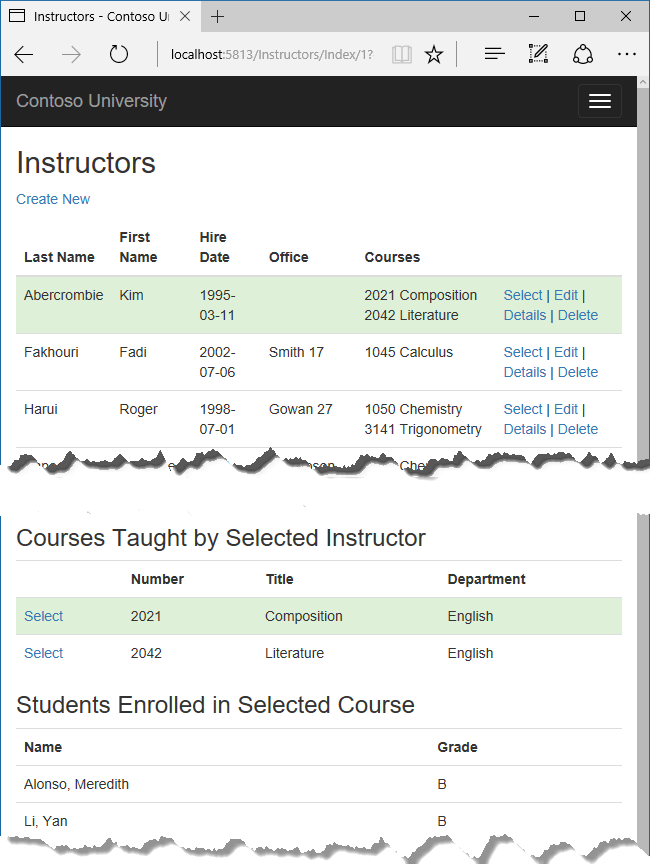
Advance to the next tutorial to begin looking at more advanced topics about expanding the data model. Along the way you'll create and apply additional migrations.

## Read related data

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.





In this tutorial, you:

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

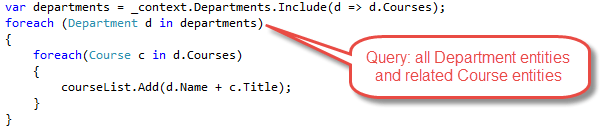
## Prerequisites

* [Create a complex data model](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/complex-data-model?view=aspnetcore-3.0)

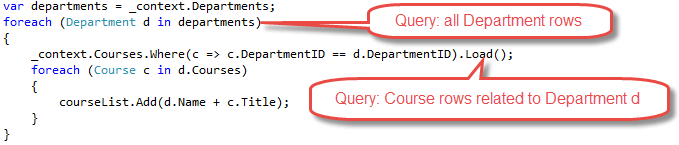
## Learn how to load related data

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:

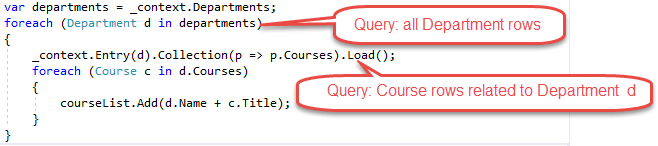
* Eager loading. When the entity is read, related data is retrieved along with it. This typically results in a single join query that retrieves all of the data that's needed. You specify eager loading in Entity Framework Core by using the Include and ThenInclude methods.



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.



* 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:



* 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

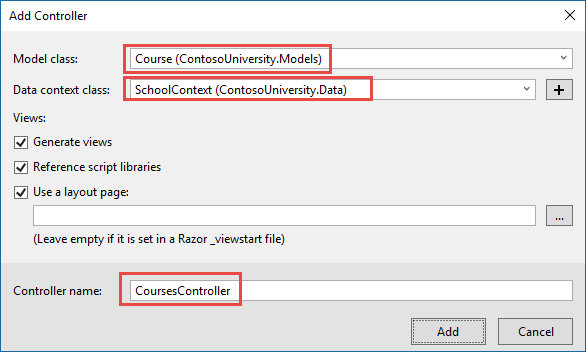
If you know you need related data for every entity retrieved, eager loading often offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. For example, suppose that each department has ten related courses. Eager 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 Students controller, as shown in the following illustration:



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

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

C#Copy

public async Task<IActionResult> Index()

{

var courses = \_context.Courses

.Include(c => c.Department)

.AsNoTracking();

return View(await courses.ToListAsync());

}

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

HTMLCopy

@model IEnumerable<ContosoUniversity.Models.Course>

@{

ViewData["Title"] = "Courses";

}

<h2>Courses</h2>

<p>

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

</p>

<table class="table">

<thead>

<tr>

<th>

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

</th>

<th>

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

</th>

<th>

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

</th>

<th>

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

</th>

<th></th>

</tr>

</thead>

<tbody>

@foreach (var item in Model)

{

<tr>

<td>

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

</td>

<td>

@Html.DisplayFor(modelItem => item.Title)

</td>

<td>

@Html.DisplayFor(modelItem => item.Credits)

</td>

<td>

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

</td>

<td>

<a asp-action="Edit" asp-route-id="@item.CourseID">Edit</a> |

<a asp-action="Details" asp-route-id="@item.CourseID">Details</a> |

<a asp-action="Delete" asp-route-id="@item.CourseID">Delete</a>

</td>

</tr>

}

</tbody>

</table>

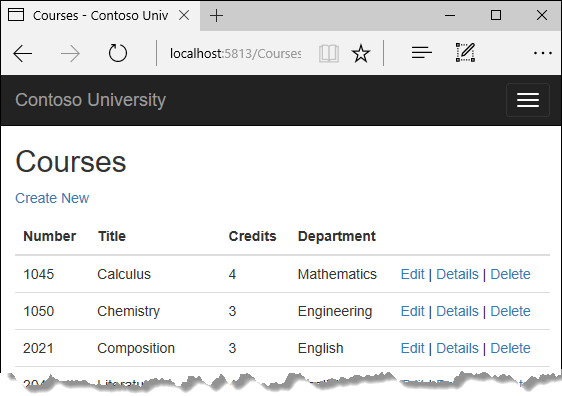
You've made the following changes to the scaffolded code:

* Changed the heading from Index to Courses.
* Added a **Number** column that shows the CourseID property value. By default, primary keys aren't scaffolded because normally they'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:

HTMLCopy

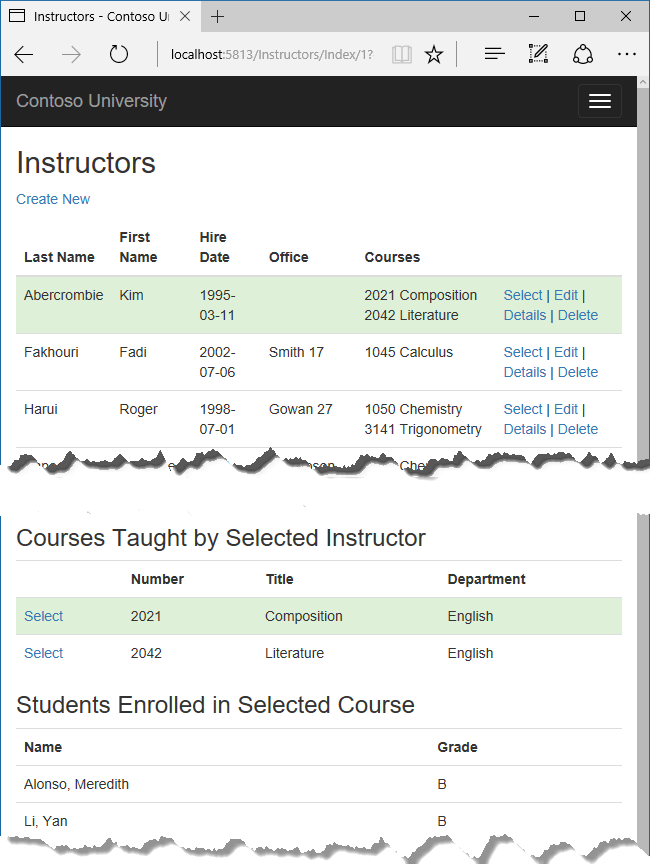
@Html.DisplayFor(modelItem => item.Department.Name)

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



## 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:



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

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

### Create a view model for the Instructor Index view

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

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

C#Copy

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels

{

public class InstructorIndexData

{

public IEnumerable<Instructor> Instructors { get; set; }

public IEnumerable<Course> Courses { get; set; }

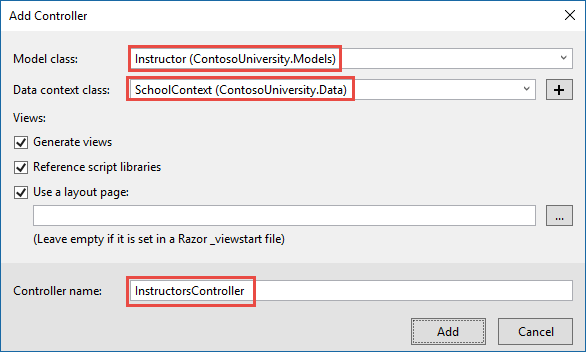
public IEnumerable<Enrollment> Enrollments { get; set; }

}

}

### Create the Instructor controller and views

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



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

C#Copy

using ContosoUniversity.Models.SchoolViewModels;

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

C#Copy

public async Task<IActionResult> Index(int? id, int? courseID)

{

var viewModel = new InstructorIndexData();

viewModel.Instructors = await \_context.Instructors

.Include(i => i.OfficeAssignment)

.Include(i => i.CourseAssignments)

.ThenInclude(i => i.Course)

.ThenInclude(i => i.Enrollments)

.ThenInclude(i => i.Student)

.Include(i => i.CourseAssignments)

.ThenInclude(i => i.Course)

.ThenInclude(i => i.Department)

.AsNoTracking()

.OrderBy(i => i.LastName)

.ToListAsync();

if (id != null)

{

ViewData["InstructorID"] = id.Value;

Instructor instructor = viewModel.Instructors.Where(

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

viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);

}

if (courseID != null)

{

ViewData["CourseID"] = courseID.Value;

viewModel.Enrollments = viewModel.Courses.Where(

x => x.CourseID == courseID).Single().Enrollments;

}

return View(viewModel);

}

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

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

C#Copy

viewModel.Instructors = await \_context.Instructors

.Include(i => i.OfficeAssignment)

.Include(i => i.CourseAssignments)

.ThenInclude(i => i.Course)

.ThenInclude(i => i.Enrollments)

.ThenInclude(i => i.Student)

.Include(i => i.CourseAssignments)

.ThenInclude(i => i.Course)

.ThenInclude(i => i.Department)

.AsNoTracking()

.OrderBy(i => i.LastName)

.ToListAsync();

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

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

C#Copy

viewModel.Instructors = await \_context.Instructors

.Include(i => i.OfficeAssignment)

.Include(i => i.CourseAssignments)

.ThenInclude(i => i.Course)

.ThenInclude(i => i.Enrollments)

.ThenInclude(i => i.Student)

.Include(i => i.CourseAssignments)

.ThenInclude(i => i.Course)

.ThenInclude(i => i.Department)

.AsNoTracking()

.OrderBy(i => i.LastName)

.ToListAsync();

At that point in the code, another ThenInclude would be for navigation properties of Student, which you don't need. But calling Include starts over with Instructor properties, so you have to go through the chain again, this time specifying Course.Department instead of Course.Enrollments.

C#Copy

viewModel.Instructors = await \_context.Instructors

.Include(i => i.OfficeAssignment)

.Include(i => i.CourseAssignments)

.ThenInclude(i => i.Course)

.ThenInclude(i => i.Enrollments)

.ThenInclude(i => i.Student)

.Include(i => i.CourseAssignments)

.ThenInclude(i => i.Course)

.ThenInclude(i => i.Department)

.AsNoTracking()

.OrderBy(i => i.LastName)

.ToListAsync();

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

C#Copy

if (id != null)

{

ViewData["InstructorID"] = id.Value;

Instructor instructor = viewModel.Instructors.Where(

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

viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);

}

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

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

C#Copy

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

Instead of:

C#Copy

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

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

C#Copy

if (courseID != null)

{

ViewData["CourseID"] = courseID.Value;

viewModel.Enrollments = viewModel.Courses.Where(

x => x.CourseID == courseID).Single().Enrollments;

}

### Modify the Instructor Index view

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

HTMLCopy

@model ContosoUniversity.Models.SchoolViewModels.InstructorIndexData

@{

ViewData["Title"] = "Instructors";

}

<h2>Instructors</h2>

<p>

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

</p>

<table class="table">

<thead>

<tr>

<th>Last Name</th>

<th>First Name</th>

<th>Hire Date</th>

<th>Office</th>

<th>Courses</th>

<th></th>

</tr>

</thead>

<tbody>

@foreach (var item in Model.Instructors)

{

string selectedRow = "";

if (item.ID == (int?)ViewData["InstructorID"])

{

selectedRow = "success";

}

<tr class="@selectedRow">

<td>

@Html.DisplayFor(modelItem => item.LastName)

</td>

<td>

@Html.DisplayFor(modelItem => item.FirstMidName)

</td>

<td>

@Html.DisplayFor(modelItem => item.HireDate)

</td>

<td>

@if (item.OfficeAssignment != null)

{

@item.OfficeAssignment.Location

}

</td>

<td>

@{

foreach (var course in item.CourseAssignments)

{

@course.Course.CourseID @: @course.Course.Title <br />

}

}

</td>

<td>

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

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

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

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

</td>

</tr>

}

</tbody>

</table>

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

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

HTMLCopy

@if (item.OfficeAssignment != null)

{

@item.OfficeAssignment.Location

}

* Added a **Courses** column that displays courses taught by each instructor. For more information, see the [Explicit line transition](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-3.0#explicit-line-transition) section of the Razor syntax article.
* Added code that dynamically adds class="success" to the tr element of the selected instructor. This sets a background color for the selected row using a Bootstrap class.

HTMLCopy

string selectedRow = "";

if (item.ID == (int?)ViewData["InstructorID"])

{

selectedRow = "success";

}

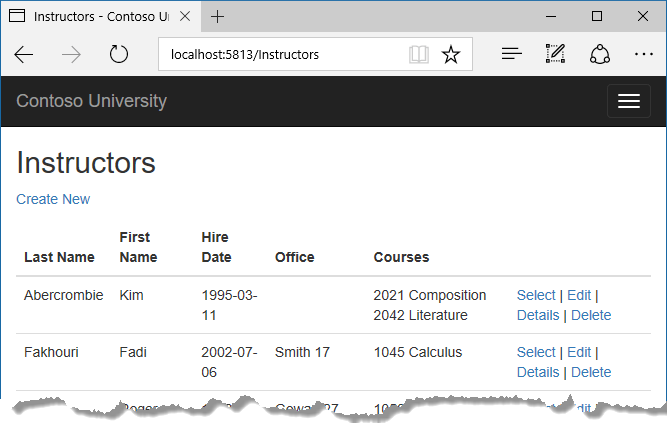
<tr class="@selectedRow">

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

HTMLCopy

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

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



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

HTMLCopy

@if (Model.Courses != null)

{

<h3>Courses Taught by Selected Instructor</h3>

<table class="table">

<tr>

<th></th>

<th>Number</th>

<th>Title</th>

<th>Department</th>

</tr>

@foreach (var item in Model.Courses)

{

string selectedRow = "";

if (item.CourseID == (int?)ViewData["CourseID"])

{

selectedRow = "success";

}

<tr class="@selectedRow">

<td>

@Html.ActionLink("Select", "Index", new { courseID = item.CourseID })

</td>

<td>

@item.CourseID

</td>

<td>

@item.Title

</td>

<td>

@item.Department.Name

</td>

</tr>

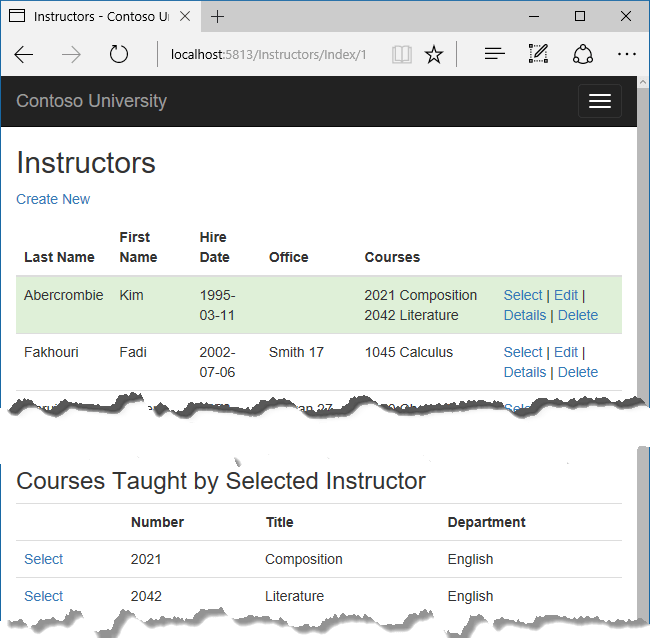
}

</table>

}

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

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



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

HTMLCopy

@if (Model.Enrollments != null)

{

<h3>

Students Enrolled in Selected Course

</h3>

<table class="table">

<tr>

<th>Name</th>

<th>Grade</th>

</tr>

@foreach (var item in Model.Enrollments)

{

<tr>

<td>

@item.Student.FullName

</td>

<td>

@Html.DisplayFor(modelItem => item.Grade)

</td>

</tr>

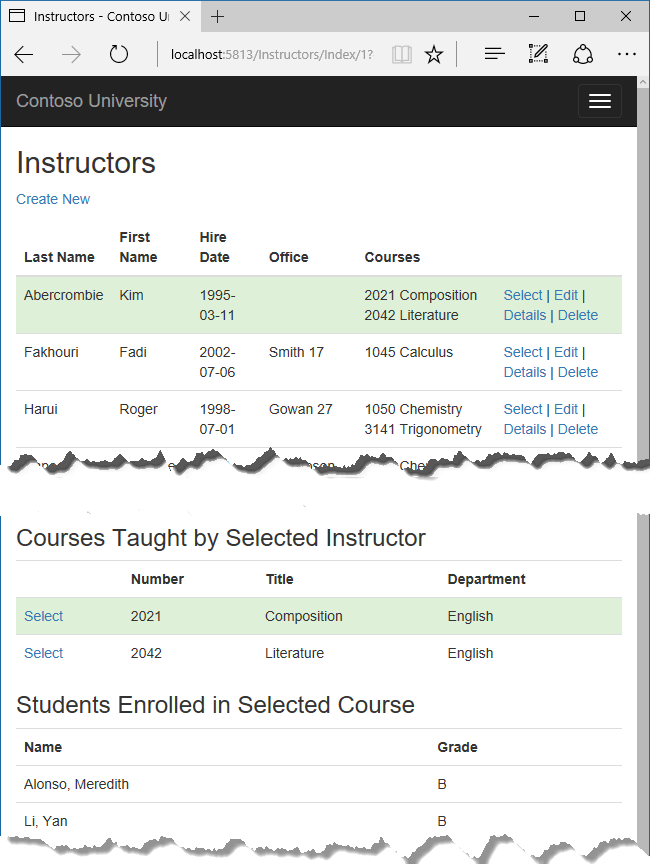
}

</table>

}

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

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



## About explicit loading

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

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

C#Copy

public async Task<IActionResult> Index(int? id, int? courseID)

{

var viewModel = new InstructorIndexData();

viewModel.Instructors = await \_context.Instructors

.Include(i => i.OfficeAssignment)

.Include(i => i.CourseAssignments)

.ThenInclude(i => i.Course)

.ThenInclude(i => i.Department)

.OrderBy(i => i.LastName)

.ToListAsync();

if (id != null)

{

ViewData["InstructorID"] = id.Value;

Instructor instructor = viewModel.Instructors.Where(

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

viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);

}

if (courseID != null)

{

ViewData["CourseID"] = courseID.Value;

var selectedCourse = viewModel.Courses.Where(x => x.CourseID == courseID).Single();

await \_context.Entry(selectedCourse).Collection(x => x.Enrollments).LoadAsync();

foreach (Enrollment enrollment in selectedCourse.Enrollments)

{

await \_context.Entry(enrollment).Reference(x => x.Student).LoadAsync();

}

viewModel.Enrollments = selectedCourse.Enrollments;

}

return View(viewModel);

}

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

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

## Next steps

In this tutorial, you:

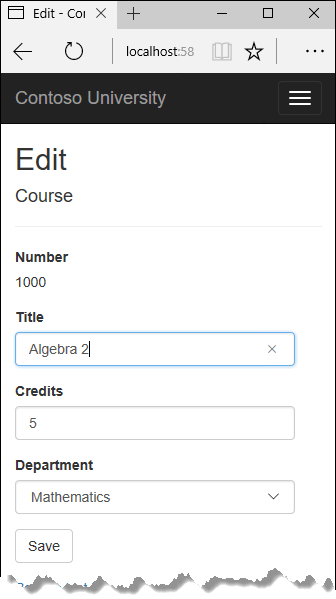
* Learned how to load related data
* Created a Courses page
* Created an Instructors page
* Learned about explicit loading

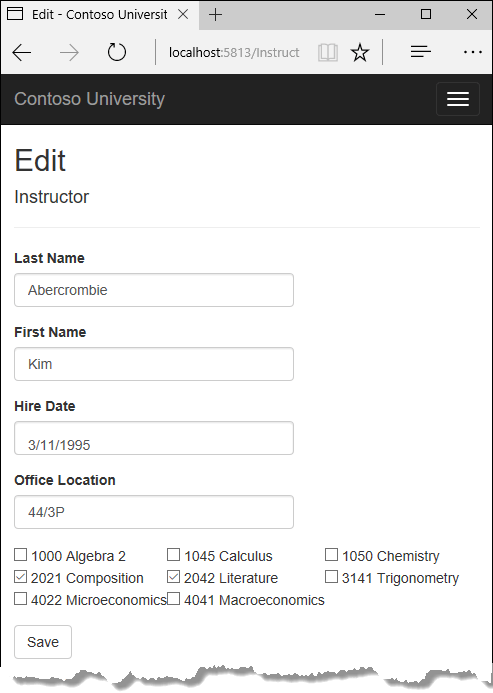
Advance to the next tutorial to learn how to update related data.

## Update related data

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.





In this tutorial, you:

* Customize Courses pages
* Add Instructors Edit page
* Add courses to Edit page
* Update Delete page
* Add office location and courses to Create page

## Prerequisites

* [Read related data](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/read-related-data?view=aspnetcore-3.0)

## Customize Courses pages

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

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

C#Copy

public IActionResult Create()

{

PopulateDepartmentsDropDownList();

return View();

}

C#Copy

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create([Bind("CourseID,Credits,DepartmentID,Title")] Course course)

{

if (ModelState.IsValid)

{

\_context.Add(course);

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

PopulateDepartmentsDropDownList(course.DepartmentID);

return View(course);

}

C#Copy

public async Task<IActionResult> Edit(int? id)

{

if (id == null)

{

return NotFound();

}

var course = await \_context.Courses

.AsNoTracking()

.FirstOrDefaultAsync(m => m.CourseID == id);

if (course == null)

{

return NotFound();

}

PopulateDepartmentsDropDownList(course.DepartmentID);

return View(course);

}

C#Copy

[HttpPost, ActionName("Edit")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> EditPost(int? id)

{

if (id == null)

{

return NotFound();

}

var courseToUpdate = await \_context.Courses

.FirstOrDefaultAsync(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.

C#Copy

private void PopulateDepartmentsDropDownList(object selectedDepartment = null)

{

var departmentsQuery = from d in \_context.Departments

orderby d.Name

select d;

ViewBag.DepartmentID = new SelectList(departmentsQuery.AsNoTracking(), "DepartmentID", "Name", selectedDepartment);

}

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

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

C#Copy

public IActionResult Create()

{

PopulateDepartmentsDropDownList();

return View();

}

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

C#Copy

public async Task<IActionResult> Edit(int? id)

{

if (id == null)

{

return NotFound();

}

var course = await \_context.Courses

.AsNoTracking()

.FirstOrDefaultAsync(m => m.CourseID == id);

if (course == null)

{

return NotFound();

}

PopulateDepartmentsDropDownList(course.DepartmentID);

return View(course);

}

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

### Add .AsNoTracking to Details and Delete methods

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

C#Copy

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

{

if (id == null)

{

return NotFound();

}

var course = await \_context.Courses

.Include(c => c.Department)

.AsNoTracking()

.FirstOrDefaultAsync(m => m.CourseID == id);

if (course == null)

{

return NotFound();

}

return View(course);

}

C#Copy

public async Task<IActionResult> Delete(int? id)

{

if (id == null)

{

return NotFound();

}

var course = await \_context.Courses

.Include(c => c.Department)

.AsNoTracking()

.FirstOrDefaultAsync(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.

HTMLCopy

<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.

HTMLCopy

<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.

HTMLCopy

@model ContosoUniversity.Models.Course

@{

ViewData["Title"] = "Delete";

}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>

<div>

<h4>Course</h4>

<hr />

<dl class="row">

<dt class="col-sm-2">

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

</dt>

<dd class="col-sm-10">

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

</dd>

<dt class="col-sm-2">

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

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.Title)

</dd>

<dt class="col-sm-2">

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

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.Credits)

</dd>

<dt class="col-sm-2">

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

</dt>

<dd class="col-sm-10">

@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>

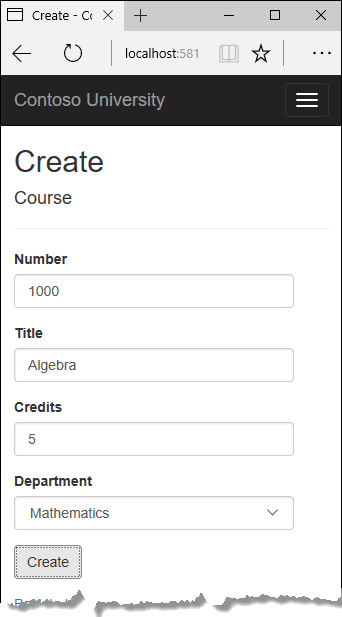
</form>

</div>

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

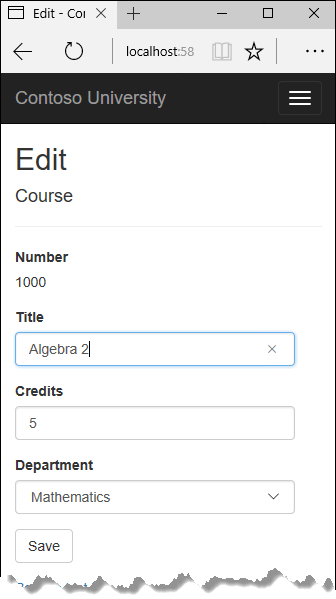
### Test the Course pages

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 Instructors Edit page

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

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

### Update the Instructors controller

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

C#Copy

public async Task<IActionResult> Edit(int? id)

{

if (id == null)

{

return NotFound();

}

var instructor = await \_context.Instructors

.Include(i => i.OfficeAssignment)

.AsNoTracking()

.FirstOrDefaultAsync(m => m.ID == id);

if (instructor == null)

{

return NotFound();

}

return View(instructor);

}

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

C#Copy

[HttpPost, ActionName("Edit")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> EditPost(int? id)

{

if (id == null)

{

return NotFound();

}

var instructorToUpdate = await \_context.Instructors

.Include(i => i.OfficeAssignment)

.FirstOrDefaultAsync(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](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/crud?view=aspnetcore-3.0).

C#Copy

if (await TryUpdateModelAsync<Instructor>(

instructorToUpdate,

"",

i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))

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

C#Copy

if (String.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:

HTMLCopy

<div class="form-group">

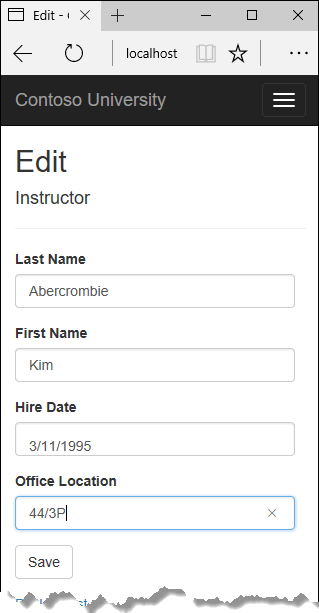
<label asp-for="OfficeAssignment.Location" class="control-label"></label>

<input asp-for="OfficeAssignment.Location" class="form-control" />

<span asp-validation-for="OfficeAssignment.Location" class="text-danger" />

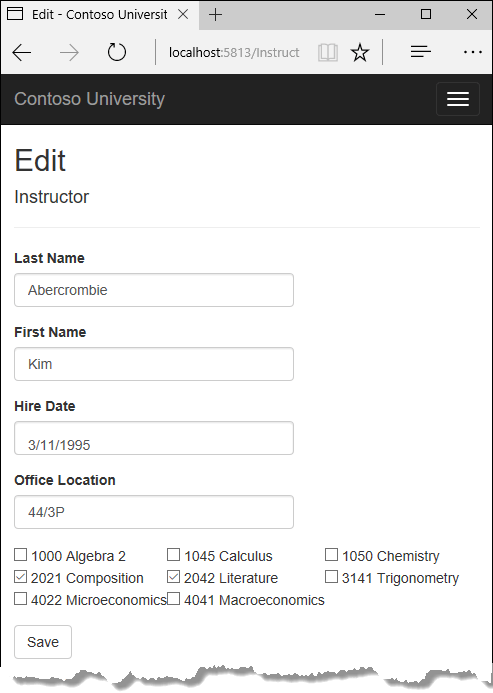
</div>

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



## Add courses to 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:

C#Copy

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels

{

public class AssignedCourseData

{

public int CourseID { get; set; }

public string Title { get; set; }

public bool Assigned { get; set; }

}

}

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

C#Copy

public async Task<IActionResult> Edit(int? id)

{

if (id == null)

{

return NotFound();

}

var instructor = await \_context.Instructors

.Include(i => i.OfficeAssignment)

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

.AsNoTracking()

.FirstOrDefaultAsync(m => m.ID == id);

if (instructor == null)

{

return NotFound();

}

PopulateAssignedCourseData(instructor);

return View(instructor);

}

private void PopulateAssignedCourseData(Instructor instructor)

{

var allCourses = \_context.Courses;

var instructorCourses = new HashSet<int>(instructor.CourseAssignments.Select(c => c.CourseID));

var viewModel = new List<AssignedCourseData>();

foreach (var course in allCourses)

{

viewModel.Add(new AssignedCourseData

{

CourseID = course.CourseID,

Title = course.Title,

Assigned = instructorCourses.Contains(course.CourseID)

});

}

ViewData["Courses"] = viewModel;

}

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

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

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

C#Copy

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(int? id, string[] selectedCourses)

{

if (id == null)

{

return NotFound();

}

var instructorToUpdate = await \_context.Instructors

.Include(i => i.OfficeAssignment)

.Include(i => i.CourseAssignments)

.ThenInclude(i => i.Course)

.FirstOrDefaultAsync(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);

}

C#Copy

private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)

{

if (selectedCourses == null)

{

instructorToUpdate.CourseAssignments = new List<CourseAssignment>();

return;

}

var selectedCoursesHS = new HashSet<string>(selectedCourses);

var instructorCourses = new HashSet<int>

(instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));

foreach (var course in \_context.Courses)

{

if (selectedCoursesHS.Contains(course.CourseID.ToString()))

{

if (!instructorCourses.Contains(course.CourseID))

{

instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID = course.CourseID });

}

}

else

{

if (instructorCourses.Contains(course.CourseID))

{

CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.FirstOrDefault(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:

C#Copy

private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)

{

if (selectedCourses == null)

{

instructorToUpdate.CourseAssignments = new List<CourseAssignment>();

return;

}

var selectedCoursesHS = new HashSet<string>(selectedCourses);

var instructorCourses = new HashSet<int>

(instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));

foreach (var course in \_context.Courses)

{

if (selectedCoursesHS.Contains(course.CourseID.ToString()))

{

if (!instructorCourses.Contains(course.CourseID))

{

instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID = course.CourseID });

}

}

else

{

if (instructorCourses.Contains(course.CourseID))

{

CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.FirstOrDefault(i => i.CourseID == course.CourseID);

\_context.Remove(courseToRemove);

}

}

}

}

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

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

C#Copy

private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)

{

if (selectedCourses == null)

{

instructorToUpdate.CourseAssignments = new List<CourseAssignment>();

return;

}

var selectedCoursesHS = new HashSet<string>(selectedCourses);

var instructorCourses = new HashSet<int>

(instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));

foreach (var course in \_context.Courses)

{

if (selectedCoursesHS.Contains(course.CourseID.ToString()))

{

if (!instructorCourses.Contains(course.CourseID))

{

instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID = course.CourseID });

}

}

else

{

if (instructorCourses.Contains(course.CourseID))

{

CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.FirstOrDefault(i => i.CourseID == course.CourseID);

\_context.Remove(courseToRemove);

}

}

}

}

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

C#Copy

private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)

{

if (selectedCourses == null)

{

instructorToUpdate.CourseAssignments = new List<CourseAssignment>();

return;

}

var selectedCoursesHS = new HashSet<string>(selectedCourses);

var instructorCourses = new HashSet<int>

(instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));

foreach (var course in \_context.Courses)

{

if (selectedCoursesHS.Contains(course.CourseID.ToString()))

{

if (!instructorCourses.Contains(course.CourseID))

{

instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID = course.CourseID });

}

}

else

{

if (instructorCourses.Contains(course.CourseID))

{

CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.FirstOrDefault(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 might be changed in a way that breaks the code. If the code looks different after pasting, 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. This problem is fixed in Visual Studio 2019.

HTMLCopy

<div class="form-group">

<div class="col-md-offset-2 col-md-10">

<table>

<tr>

@{

int cnt = 0;

List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses = ViewBag.Courses;

foreach (var course in courses)

{

if (cnt++ % 3 == 0)

{

@:</tr><tr>

}

@:<td>

<input type="checkbox"

name="selectedCourses"

value="@course.CourseID"

@(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />

@course.CourseID @: @course.Title

@:</td>

}

@:</tr>

}

</table>

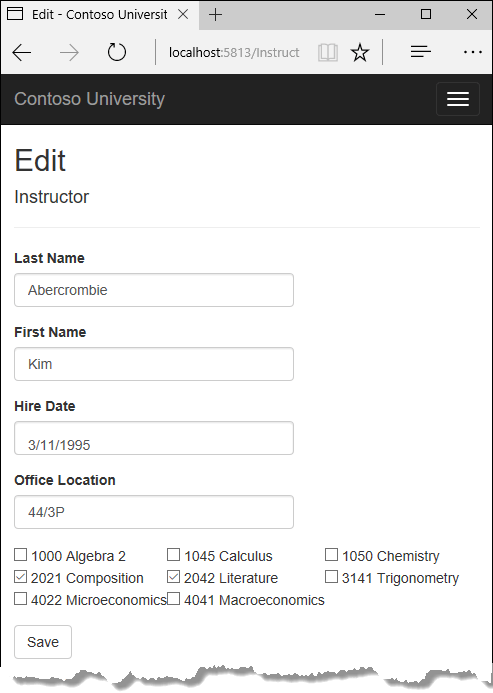
</div>

</div>

This code creates an HTML table that has three columns. In each column is a check box followed by a caption that consists of the course number and title. The check boxes all have the same name ("selectedCourses"), which informs the model binder that they'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 Delete page

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

C#Copy

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeleteConfirmed(int id)

{

Instructor instructor = await \_context.Instructors

.Include(i => i.CourseAssignments)

.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 Create page

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

C#Copy

public IActionResult Create()

{

var instructor = new Instructor();

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

PopulateAssignedCourseData(instructor);

return View();

}

// POST: Instructors/Create

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Create([Bind("FirstMidName,HireDate,LastName,OfficeAssignment")] Instructor instructor, string[] selectedCourses)

{

if (selectedCourses != null)

{

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

foreach (var course in selectedCourses)

{

var courseToAdd = new CourseAssignment { InstructorID = instructor.ID, CourseID = int.Parse(course) };

instructor.CourseAssignments.Add(courseToAdd);

}

}

if (ModelState.IsValid)

{

\_context.Add(instructor);

await \_context.SaveChangesAsync();

return RedirectToAction(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:

C#Copy

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

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

C#Copy

private ICollection<CourseAssignment> \_courseAssignments;

public ICollection<CourseAssignment> CourseAssignments

{

get

{

return \_courseAssignments ?? (\_courseAssignments = new List<CourseAssignment>());

}

set

{

\_courseAssignments = value;

}

}

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

In Views/Instructor/Create.cshtml, add an office location text box and check boxes for courses before the Submit button. As in the case of the Edit page, [fix the formatting if Visual Studio reformats the code when you paste it](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/update-related-data?view=aspnetcore-3.0#notepad).

HTMLCopy

<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](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/crud?view=aspnetcore-3.0), 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](https://docs.microsoft.com/en-us/ef/core/saving/transactions).

## Next steps

In this tutorial, you:

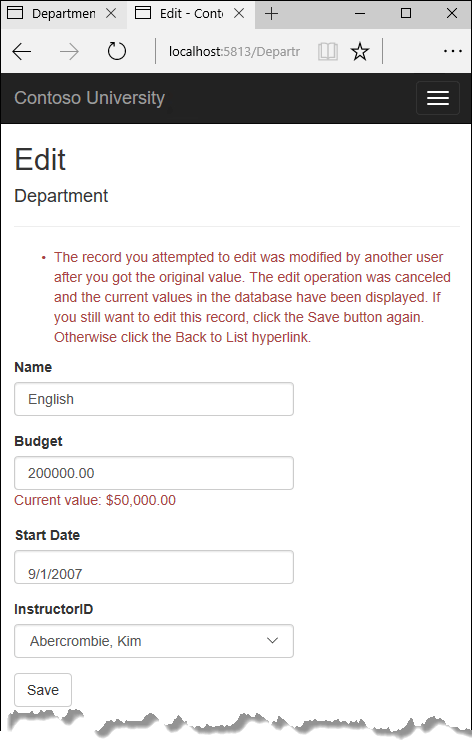
* Customized Courses pages
* Added Instructors Edit page
* Added courses to Edit page
* Updated Delete page
* Added office location and courses to Create page

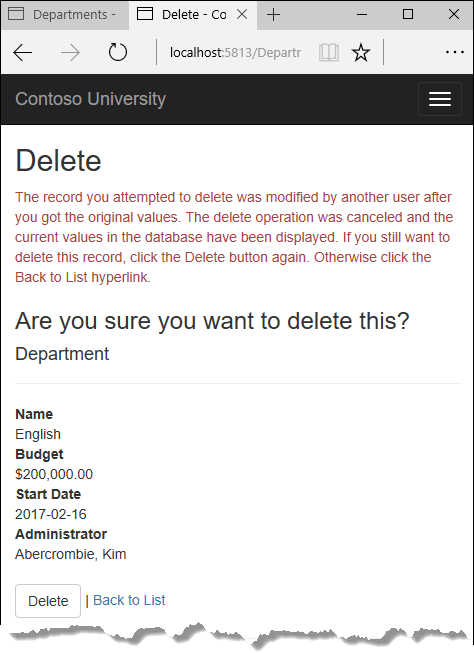
Advance to the next tutorial to learn how to handle concurrency conflicts.

## Handle concurrency conflicts

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.





In this tutorial, you:

* Learn about concurrency conflicts
* Add a tracking property
* Create Departments controller and views
* Update Index view
* Update Edit methods
* Update Edit view
* Test concurrency conflicts
* Update the Delete page
* Update Details and Create views

## Prerequisites

* [Update related data](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/update-related-data?view=aspnetcore-3.0)

## Concurrency conflicts

A concurrency conflict occurs when one user displays an entity's data in order to edit it, and then another user updates the same entity's data before the first user's change is written to the database. If you don't enable the detection of such conflicts, whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable: if there are few users, or few updates, or if isn't really critical if some changes are overwritten, the cost of programming for concurrency might outweigh the benefit. In that case, you don't have to configure the application to handle concurrency conflicts.

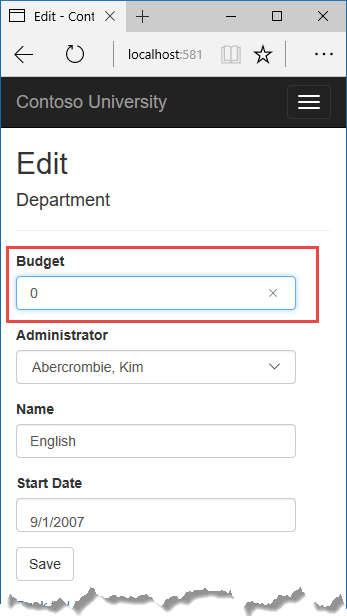
### Pessimistic concurrency (locking)

If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called pessimistic concurrency. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

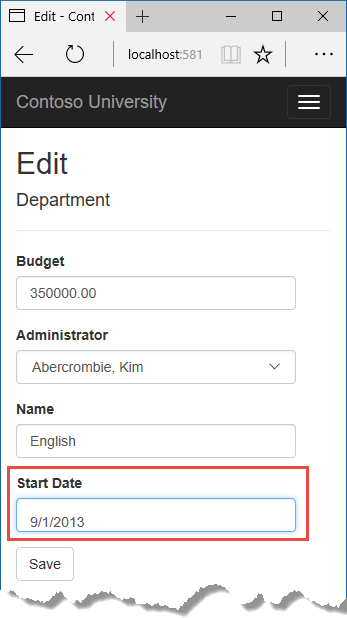
Managing locks has disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases. For these reasons, not all database management systems support pessimistic concurrency. Entity Framework Core provides no built-in support for it, and this tutorial doesn't show you how to implement it.

### Optimistic Concurrency

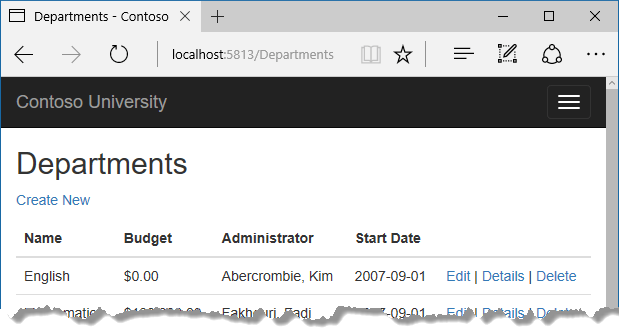
The alternative to pessimistic concurrency is optimistic concurrency. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, Jane visits the Department Edit page and changes the Budget amount for the English department from $350,000.00 to $0.00.



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



Jane clicks **Save** first and sees her change when the browser returns to the Index page.



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

Some of the options include the following:

* You can keep track of which property a user has modified and update only the corresponding columns in the database.

In the example scenario, no data would be lost, because different properties were updated by the two users. The next time someone browses the English department, they will see both Jane's and John's changes -- a start date of 9/1/2013 and a budget of zero dollars. This method of updating can reduce the number of conflicts that could result in data loss, but it can't avoid data loss if competing changes are made to the same property of an entity. Whether the Entity Framework works this way depends on how you implement your update code. It's often not practical in a web application, because it can require that you maintain large amounts of state in order to keep track of all original property values for an entity as well as new values. Maintaining large amounts of state can affect application performance because it either requires server resources or must be included in the web page itself (for example, in hidden fields) or in a cookie.

* You can let John's change overwrite Jane's change.

The next time someone browses the English department, they will see 9/1/2013 and the restored $350,000.00 value. This is called a Client Wins or Last in Wins scenario. (All values from the client take precedence over what's in the data store.) As noted in the introduction to this section, if you don't do any coding for concurrency handling, this will happen automatically.

* You can prevent John's change from being updated in the database.

Typically, you would display an error message, show him the current state of the data, and allow him to reapply his changes if he still wants to make them. This is called a Store Wins scenario. (The data-store values take precedence over the values submitted by the client.) You'll implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted to what's happening.

### Detecting concurrency conflicts

You can resolve conflicts by handling DbConcurrencyException exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database and the data model appropriately. Some options for enabling conflict detection include the following:

* In the database table, include a tracking column that can be used to determine when a row has been changed. You can then configure the Entity Framework to include that column in the Where clause of SQL Update or Delete commands.

The data type of the tracking column is typically rowversion. The rowversion value is a sequential number that's incremented each time the row is updated. In an Update or Delete command, the Where clause includes the original value of the tracking column (the original row version) . If the row being updated has been changed by another user, the value in the rowversion column is different than the original value, so the Update or Delete statement can't find the row to update because of the Where clause. When the Entity Framework finds that no rows have been updated by the Update or Delete command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

* Configure the Entity Framework to include the original values of every column in the table in the Where clause of Update and Delete commands.

As in the first option, if anything in the row has changed since the row was first read, the Where clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. For database tables that have many columns, this approach can result in very large Where clauses, and can require that you maintain large amounts of state. As noted earlier, maintaining large amounts of state can affect application performance. Therefore this approach is generally not recommended, and it isn't the method used in this tutorial.

If you do want to implement this approach to concurrency, you have to mark all non-primary-key properties in the entity you want to track concurrency for by adding the ConcurrencyCheck attribute to them. That change enables the Entity Framework to include all columns in the SQL Where clause of Update and Delete statements.

In the remainder of this tutorial you'll add a rowversion tracking property to the Department entity, create a controller and views, and test to verify that everything works correctly.

## Add a tracking property

In Models/Department.cs, add a tracking property named RowVersion:

C#Copy

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models

{

public class Department

{

public int DepartmentID { get; set; }

[StringLength(50, MinimumLength = 3)]

public string Name { get; set; }

[DataType(DataType.Currency)]

[Column(TypeName = "money")]

public decimal Budget { get; set; }

[DataType(DataType.Date)]

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

[Display(Name = "Start Date")]

public DateTime StartDate { get; set; }

public int? InstructorID { get; set; }

[Timestamp]

public byte[] RowVersion { get; set; }

public Instructor Administrator { get; set; }

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

}

}

The Timestamp attribute specifies that this column will be included in the Where clause of Update and Delete commands sent to the database. The attribute is called Timestamp because previous versions of SQL Server used a SQL timestamp data type before the SQL rowversion replaced it. The .NET type for rowversion is a byte array.

If you prefer to use the fluent API, you can use the IsConcurrencyToken method (in Data/SchoolContext.cs) to specify the tracking property, as shown in the following example:

C#Copy

modelBuilder.Entity<Department>()

.Property(p => p.RowVersion).IsConcurrencyToken();

By adding a property you changed the database model, so you need to do another migration.

Save your changes and build the project, and then enter the following commands in the command window:

.NET Core CLICopy

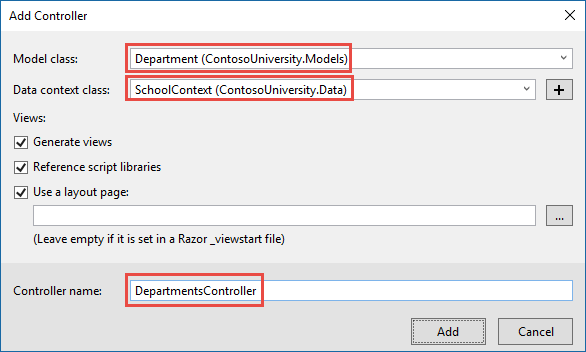
dotnet ef migrations add RowVersion

.NET Core CLICopy

dotnet ef database update

## Create Departments controller and views

Scaffold a Departments controller and views as you did earlier for Students, Courses, and Instructors.



In the DepartmentsController.cs file, change all four occurrences of "FirstMidName" to "FullName" so that the department administrator drop-down lists will contain the full name of the instructor rather than just the last name.

C#Copy

ViewData["InstructorID"] = new SelectList(\_context.Instructors, "ID", "FullName", department.InstructorID);

## Update Index view

The scaffolding engine created a RowVersion column in the Index view, but that field shouldn't be displayed.

Replace the code in Views/Departments/Index.cshtml with the following code.

HTMLCopy

@model IEnumerable<ContosoUniversity.Models.Department>

@{

ViewData["Title"] = "Departments";

}

<h2>Departments</h2>

<p>

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

</p>

<table class="table">

<thead>

<tr>

<th>

@Html.DisplayNameFor(model => model.Name)

</th>

<th>

@Html.DisplayNameFor(model => model.Budget)

</th>

<th>

@Html.DisplayNameFor(model => model.StartDate)

</th>

<th>

@Html.DisplayNameFor(model => model.Administrator)

</th>

<th></th>

</tr>

</thead>

<tbody>

@foreach (var item in Model)

{

<tr>

<td>

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

</td>

<td>

@Html.DisplayFor(modelItem => item.Budget)

</td>

<td>

@Html.DisplayFor(modelItem => item.StartDate)

</td>

<td>

@Html.DisplayFor(modelItem => item.Administrator.FullName)

</td>

<td>

<a asp-action="Edit" asp-route-id="@item.DepartmentID">Edit</a> |

<a asp-action="Details" asp-route-id="@item.DepartmentID">Details</a> |

<a asp-action="Delete" asp-route-id="@item.DepartmentID">Delete</a>

</td>

</tr>

}

</tbody>

</table>

This changes the heading to "Departments", deletes the RowVersion column, and shows full name instead of first name for the administrator.

## Update Edit methods

In both the HttpGet Edit method and the Details method, add AsNoTracking. In the HttpGet Edit method, add eager loading for the Administrator.

C#Copy

var department = await \_context.Departments

.Include(i => i.Administrator)

.AsNoTracking()

.FirstOrDefaultAsync(m => m.DepartmentID == id);

Replace the existing code for the HttpPost Edit method with the following code:

C#Copy

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(int? id, byte[] rowVersion)

{

if (id == null)

{

return NotFound();

}

var departmentToUpdate = await \_context.Departments.Include(i => i.Administrator).FirstOrDefaultAsync(m => m.DepartmentID == id);

if (departmentToUpdate == null)

{

Department deletedDepartment = new Department();

await TryUpdateModelAsync(deletedDepartment);

ModelState.AddModelError(string.Empty,

"Unable to save changes. The department was deleted by another user.");

ViewData["InstructorID"] = new SelectList(\_context.Instructors, "ID", "FullName", deletedDepartment.InstructorID);

return View(deletedDepartment);

}

\_context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;

if (await TryUpdateModelAsync<Department>(

departmentToUpdate,

"",

s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))

{

try

{

await \_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

catch (DbUpdateConcurrencyException ex)

{

var exceptionEntry = ex.Entries.Single();

var clientValues = (Department)exceptionEntry.Entity;

var databaseEntry = exceptionEntry.GetDatabaseValues();

if (databaseEntry == null)

{

ModelState.AddModelError(string.Empty,

"Unable to save changes. The department was deleted by another user.");

}

else

{

var databaseValues = (Department)databaseEntry.ToObject();

if (databaseValues.Name != clientValues.Name)

{

ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");

}

if (databaseValues.Budget != clientValues.Budget)

{

ModelState.AddModelError("Budget", $"Current value: {databaseValues.Budget:c}");

}

if (databaseValues.StartDate != clientValues.StartDate)

{

ModelState.AddModelError("StartDate", $"Current value: {databaseValues.StartDate:d}");

}

if (databaseValues.InstructorID != clientValues.InstructorID)

{

Instructor databaseInstructor = await \_context.Instructors.FirstOrDefaultAsync(i => i.ID == databaseValues.InstructorID);

ModelState.AddModelError("InstructorID", $"Current value: {databaseInstructor?.FullName}");

}

ModelState.AddModelError(string.Empty, "The record you attempted to edit "

+ "was modified by another user after you got the original value. The "

+ "edit operation was canceled and the current values in the database "

+ "have been displayed. If you still want to edit this record, click "

+ "the Save button again. Otherwise click the Back to List hyperlink.");

departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;

ModelState.Remove("RowVersion");

}

}

}

ViewData["InstructorID"] = new SelectList(\_context.Instructors, "ID", "FullName", departmentToUpdate.InstructorID);

return View(departmentToUpdate);

}

The code begins by trying to read the department to be updated. If the FirstOrDefaultAsync method returns null, the department was deleted by another user. In that case the code uses the posted form values to create a department entity so that the Edit page can be redisplayed with an error message. As an alternative, you wouldn't have to re-create the department entity if you display only an error message without redisplaying the department fields.

The view stores the original RowVersion value in a hidden field, and this method receives that value in the rowVersion parameter. Before you call SaveChanges, you have to put that original RowVersion property value in the OriginalValues collection for the entity.

C#Copy

\_context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;

Then when the Entity Framework creates a SQL UPDATE command, that command will include a WHERE clause that looks for a row that has the original RowVersion value. If no rows are affected by the UPDATE command (no rows have the original RowVersion value), the Entity Framework throws a DbUpdateConcurrencyException exception.

The code in the catch block for that exception gets the affected Department entity that has the updated values from the Entries property on the exception object.

C#Copy

var exceptionEntry = ex.Entries.Single();

The Entries collection will have just one EntityEntry object. You can use that object to get the new values entered by the user and the current database values.

C#Copy

var clientValues = (Department)exceptionEntry.Entity;

var databaseEntry = exceptionEntry.GetDatabaseValues();

The code adds a custom error message for each column that has database values different from what the user entered on the Edit page (only one field is shown here for brevity).

C#Copy

var databaseValues = (Department)databaseEntry.ToObject();

if (databaseValues.Name != clientValues.Name)

{

ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");

Finally, the code sets the RowVersion value of the departmentToUpdate to the new value retrieved from the database. This new RowVersion value will be stored in the hidden field when the Edit page is redisplayed, and the next time the user clicks **Save**, only concurrency errors that happen since the redisplay of the Edit page will be caught.

C#Copy

departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;

ModelState.Remove("RowVersion");

The ModelState.Remove statement is required because ModelState has the old RowVersion value. In the view, the ModelState value for a field takes precedence over the model property values when both are present.

## Update Edit view

In Views/Departments/Edit.cshtml, make the following changes:

* Add a hidden field to save the RowVersion property value, immediately following the hidden field for the DepartmentID property.
* Add a "Select Administrator" option to the drop-down list.

HTMLCopy

@model ContosoUniversity.Models.Department

@{

ViewData["Title"] = "Edit";

}

<h2>Edit</h2>

<h4>Department</h4>

<hr />

<div class="row">

<div class="col-md-4">

<form asp-action="Edit">

<div asp-validation-summary="ModelOnly" class="text-danger"></div>

<input type="hidden" asp-for="DepartmentID" />

<input type="hidden" asp-for="RowVersion" />

<div class="form-group">

<label asp-for="Name" class="control-label"></label>

<input asp-for="Name" class="form-control" />

<span asp-validation-for="Name" class="text-danger"></span>

</div>

<div class="form-group">

<label asp-for="Budget" class="control-label"></label>

<input asp-for="Budget" class="form-control" />

<span asp-validation-for="Budget" class="text-danger"></span>

</div>

<div class="form-group">

<label asp-for="StartDate" class="control-label"></label>

<input asp-for="StartDate" class="form-control" />

<span asp-validation-for="StartDate" class="text-danger"></span>

</div>

<div class="form-group">

<label asp-for="InstructorID" class="control-label"></label>

<select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID">

<option value="">-- Select Administrator --</option>

</select>

<span asp-validation-for="InstructorID" class="text-danger"></span>

</div>

<div class="form-group">

<input type="submit" value="Save" class="btn btn-default" />

</div>

</form>

</div>

</div>

<div>

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

</div>

@section Scripts {

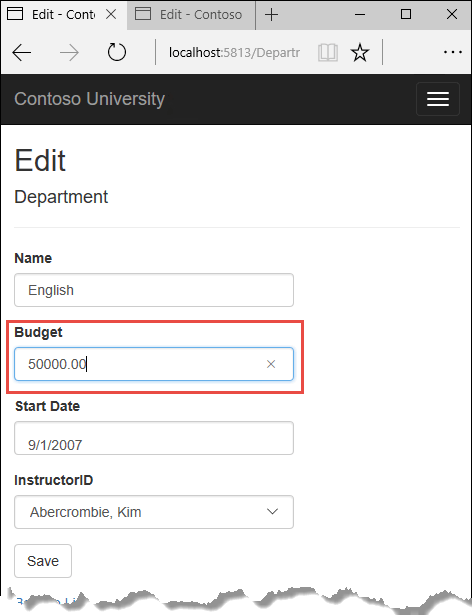
@{await Html.RenderPartialAsync("\_ValidationScriptsPartial");}

}

## Test concurrency conflicts

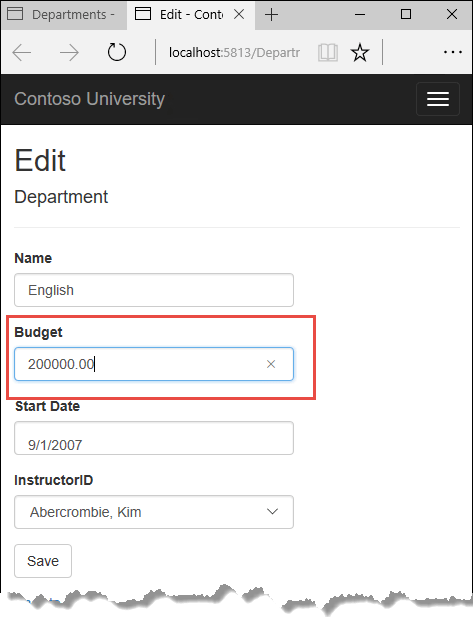
Run the app and go to the Departments Index page. Right-click the **Edit** hyperlink for the English department and select **Open in new tab**, then click the **Edit** hyperlink for the English department. The two browser tabs now display the same information.

Change a field in the first browser tab and click **Save**.

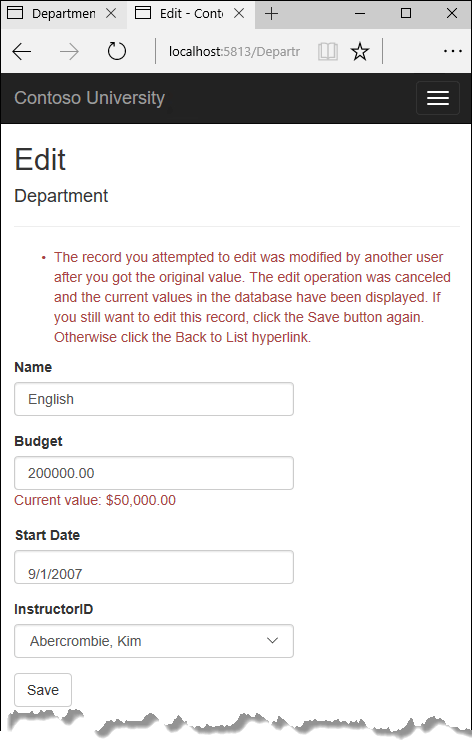


The browser shows the Index page with the changed value.

Change a field in the second browser tab.



Click **Save**. You see an error message:



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

## Update the Delete page

For the Delete page, the Entity Framework detects concurrency conflicts caused by someone else editing the department in a similar manner. When the HttpGet Delete method displays the confirmation view, the view includes the original RowVersion value in a hidden field. That value is then available to the HttpPost Delete method that's called when the user confirms the deletion. When the Entity Framework creates the SQL DELETE command, it includes a WHERE clause with the original RowVersion value. If the command results in zero rows affected (meaning the row was changed after the Delete confirmation page was displayed), a concurrency exception is thrown, and the HttpGet Delete method is called with an error flag set to true in order to redisplay the confirmation page with an error message. It's also possible that zero rows were affected because the row was deleted by another user, so in that case no error message is displayed.

### Update the Delete methods in the Departments controller

In DepartmentsController.cs, replace the HttpGet Delete method with the following code:

C#Copy

public async Task<IActionResult> Delete(int? id, bool? concurrencyError)

{

if (id == null)

{

return NotFound();

}

var department = await \_context.Departments

.Include(d => d.Administrator)

.AsNoTracking()

.FirstOrDefaultAsync(m => m.DepartmentID == id);

if (department == null)

{

if (concurrencyError.GetValueOrDefault())

{

return RedirectToAction(nameof(Index));

}

return NotFound();

}

if (concurrencyError.GetValueOrDefault())

{

ViewData["ConcurrencyErrorMessage"] = "The record you attempted to delete "

+ "was modified by another user after you got the original values. "

+ "The delete operation was canceled and the current values in the "

+ "database have been displayed. If you still want to delete this "

+ "record, click the Delete button again. Otherwise "

+ "click the Back to List hyperlink.";

}

return View(department);

}

The method accepts an optional parameter that indicates whether the page is being redisplayed after a concurrency error. If this flag is true and the department specified no longer exists, it was deleted by another user. In that case, the code redirects to the Index page. If this flag is true and the Department does exist, it was changed by another user. In that case, the code sends an error message to the view using ViewData.

Replace the code in the HttpPost Delete method (named DeleteConfirmed) with the following code:

C#Copy

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Delete(Department department)

{

try

{

if (await \_context.Departments.AnyAsync(m => m.DepartmentID == department.DepartmentID))

{

\_context.Departments.Remove(department);

await \_context.SaveChangesAsync();

}

return RedirectToAction(nameof(Index));

}

catch (DbUpdateConcurrencyException /\* ex \*/)

{

//Log the error (uncomment ex variable name and write a log.)

return RedirectToAction(nameof(Delete), new { concurrencyError = true, id = department.DepartmentID });

}

}

In the scaffolded code that you just replaced, this method accepted only a record ID:

C#Copy

public async Task<IActionResult> DeleteConfirmed(int id)

You've changed this parameter to a Department entity instance created by the model binder. This gives EF access to the RowVersion property value in addition to the record key.

C#Copy

public async Task<IActionResult> Delete(Department department)

You have also changed the action method name from DeleteConfirmed to Delete. The scaffolded code used the name DeleteConfirmed to give the HttpPost method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the HttpPost and HttpGet delete methods.

If the department is already deleted, the AnyAsync method returns false and the application just goes back to the Index method.

If a concurrency error is caught, the code redisplays the Delete confirmation page and provides a flag that indicates it should display a concurrency error message.

### Update the Delete view

In Views/Departments/Delete.cshtml, replace the scaffolded code with the following code that adds an error message field and hidden fields for the DepartmentID and RowVersion properties. The changes are highlighted.

HTMLCopy

@model ContosoUniversity.Models.Department

@{

ViewData["Title"] = "Delete";

}

<h2>Delete</h2>

<p class="text-danger">@ViewData["ConcurrencyErrorMessage"]</p>

<h3>Are you sure you want to delete this?</h3>

<div>

<h4>Department</h4>

<hr />

<dl class="row">

<dt class="col-sm-2">

@Html.DisplayNameFor(model => model.Name)

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.Name)

</dd>

<dt class="col-sm-2">

@Html.DisplayNameFor(model => model.Budget)

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.Budget)

</dd>

<dt class="col-sm-2">

@Html.DisplayNameFor(model => model.StartDate)

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.StartDate)

</dd>

<dt class="col-sm-2">

@Html.DisplayNameFor(model => model.Administrator)

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.Administrator.FullName)

</dd>

</dl>

<form asp-action="Delete">

<input type="hidden" asp-for="DepartmentID" />

<input type="hidden" asp-for="RowVersion" />

<div class="form-actions no-color">

<input type="submit" value="Delete" class="btn btn-default" /> |

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

</div>

</form>

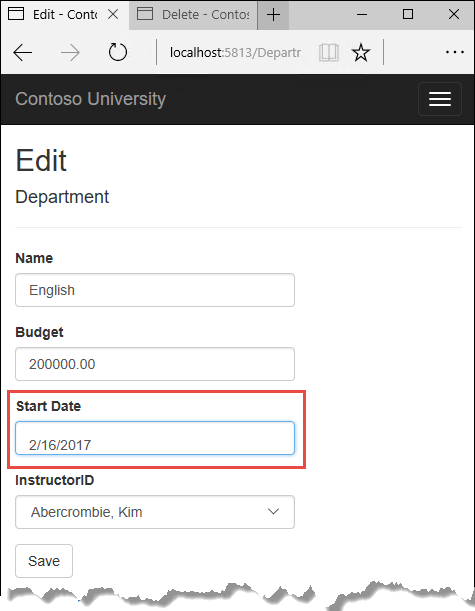
</div>

This makes the following changes:

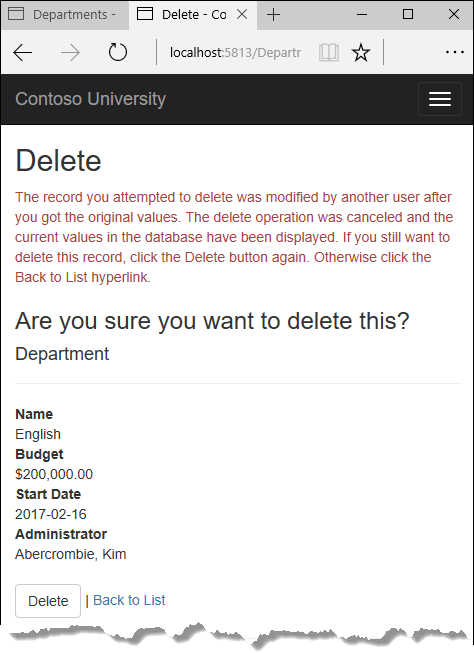
* Adds an error message between the h2 and h3 headings.
* Replaces FirstMidName with FullName in the **Administrator** field.
* Removes the RowVersion field.
* Adds a hidden field for the RowVersion property.

Run the app and go to the Departments Index page. Right-click the **Delete** hyperlink for the English department and select **Open in new tab**, then in the first tab click the **Edit** hyperlink for the English department.

In the first window, change one of the values, and click **Save**:



In the second tab, click **Delete**. You see the concurrency error message, and the Department values are refreshed with what's currently in the database.



If you click **Delete** again, you're redirected to the Index page, which shows that the department has been deleted.

## Update Details and Create views

You can optionally clean up scaffolded code in the Details and Create views.

Replace the code in Views/Departments/Details.cshtml to delete the RowVersion column and show the full name of the Administrator.

HTMLCopy

@model ContosoUniversity.Models.Department

@{

ViewData["Title"] = "Details";

}

<h2>Details</h2>

<div>

<h4>Department</h4>

<hr />

<dl class="row">

<dt class="col-sm-2">

@Html.DisplayNameFor(model => model.Name)

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.Name)

</dd>

<dt class="col-sm-2">

@Html.DisplayNameFor(model => model.Budget)

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.Budget)

</dd>

<dt class="col-sm-2">

@Html.DisplayNameFor(model => model.StartDate)

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.StartDate)

</dd>

<dt class="col-sm-2">

@Html.DisplayNameFor(model => model.Administrator)

</dt>

<dd class="col-sm-10">

@Html.DisplayFor(model => model.Administrator.FullName)

</dd>

</dl>

</div>

<div>

<a asp-action="Edit" asp-route-id="@Model.DepartmentID">Edit</a> |

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

</div>

Replace the code in Views/Departments/Create.cshtml to add a Select option to the drop-down list.

HTMLCopy

@model ContosoUniversity.Models.Department

@{

ViewData["Title"] = "Create";

}

<h2>Create</h2>

<h4>Department</h4>

<hr />

<div class="row">

<div class="col-md-4">

<form asp-action="Create">

<div asp-validation-summary="ModelOnly" class="text-danger"></div>

<div class="form-group">

<label asp-for="Name" class="control-label"></label>

<input asp-for="Name" class="form-control" />

<span asp-validation-for="Name" class="text-danger"></span>

</div>

<div class="form-group">

<label asp-for="Budget" class="control-label"></label>

<input asp-for="Budget" class="form-control" />

<span asp-validation-for="Budget" class="text-danger"></span>

</div>

<div class="form-group">

<label asp-for="StartDate" class="control-label"></label>

<input asp-for="StartDate" class="form-control" />

<span asp-validation-for="StartDate" class="text-danger"></span>

</div>

<div class="form-group">

<label asp-for="InstructorID" class="control-label"></label>

<select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID">

<option value="">-- Select Administrator --</option>

</select>

</div>

<div class="form-group">

<input type="submit" value="Create" class="btn btn-default" />

</div>

</form>

</div>

</div>

<div>

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

</div>

@section Scripts {

@{await Html.RenderPartialAsync("\_ValidationScriptsPartial");}

}

## Additional resources

For more information about how to handle concurrency in EF Core, see [Concurrency conflicts](https://docs.microsoft.com/en-us/ef/core/saving/concurrency).

## Next steps

In this tutorial, you:

* Learned about concurrency conflicts
* Added a tracking property
* Created Departments controller and views
* Updated Index view
* Updated Edit methods
* Updated Edit view
* Tested concurrency conflicts
* Updated the Delete page
* Updated Details and Create views

Advance to the next tutorial to learn how to implement table-per-hierarchy inheritance for the Instructor and Student entities.

## Inheritance

In the previous tutorial, you handled concurrency exceptions. This tutorial will show you how to implement inheritance in the data model.

In object-oriented programming, you can use inheritance to facilitate code reuse. In this tutorial, you'll change the Instructor and Student classes so that they derive from a Person base class which contains properties such as LastName that are common to both instructors and students. You won't add or change any web pages, but you'll change some of the code and those changes will be automatically reflected in the database.

In this tutorial, you:

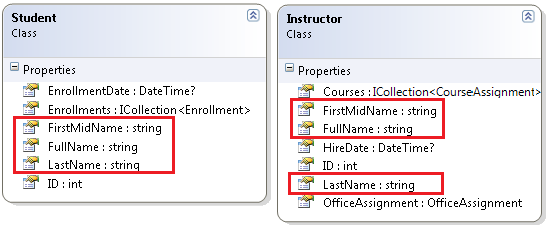
* Map inheritance to database
* Create the Person class
* Update Instructor and Student
* Add Person to the model
* Create and update migrations
* Test the implementation

## Prerequisites

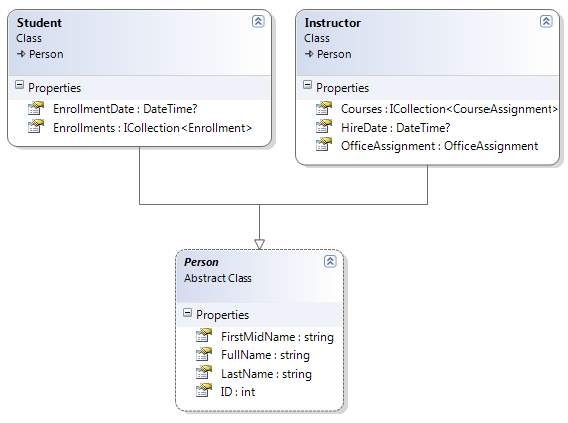
* [Handle Concurrency](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/concurrency?view=aspnetcore-3.0)

## Map inheritance to database

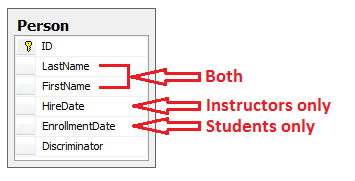
The Instructor and Student classes in the School data model have several properties that are identical:



Suppose you want to eliminate the redundant code for the properties that are shared by the Instructor and Student entities. Or you want to write a service that can format names without caring whether the name came from an instructor or a student. You could create a Person base class that contains only those shared properties, then make the Instructor and Student classes inherit from that base class, as shown in the following illustration:

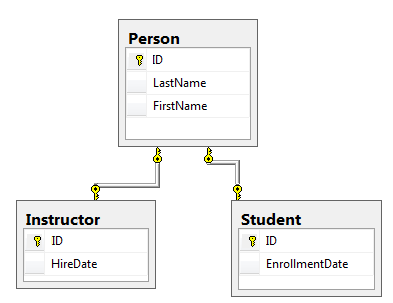


There are several ways this inheritance structure could be represented in the database. You could have a Person table that includes information about both students and instructors in a single table. Some of the columns could apply only to instructors (HireDate), some only to students (EnrollmentDate), some to both (LastName, FirstName). Typically, you'd have a discriminator column to indicate which type each row represents. For example, the discriminator column might have "Instructor" for instructors and "Student" for students.



This pattern of generating an entity inheritance structure from a single database table is called table-per-hierarchy (TPH) inheritance.

An alternative is to make the database look more like the inheritance structure. For example, you could have only the name fields in the Person table and have separate Instructor and Student tables with the date fields.



This pattern of making a database table for each entity class is called table per type (TPT) inheritance.

Yet another option is to map all non-abstract types to individual tables. All properties of a class, including inherited properties, map to columns of the corresponding table. This pattern is called Table-per-Concrete Class (TPC) inheritance. If you implemented TPC inheritance for the Person, Student, and Instructor classes as shown earlier, the Student and Instructor tables would look no different after implementing inheritance than they did before.

TPC and TPH inheritance patterns generally deliver better performance than TPT inheritance patterns, because TPT patterns can result in complex join queries.

This tutorial demonstrates how to implement TPH inheritance. TPH is the only inheritance pattern that the Entity Framework Core supports. What you'll do is create a Person class, change the Instructor and Student classes to derive from Person, add the new class to the DbContext, and create a migration.

**Tip**

Consider saving a copy of the project before making the following changes. Then if you run into problems and need to start over, it will be easier to start from the saved project instead of reversing steps done for this tutorial or going back to the beginning of the whole series.

## Create the Person class

In the Models folder, create Person.cs and replace the template code with the following code:

C#Copy

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models

{

public abstract class Person

{

public int ID { get; set; }

[Required]

[StringLength(50)]

[Display(Name = "Last Name")]

public string LastName { get; set; }

[Required]

[StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]

[Column("FirstName")]

[Display(Name = "First Name")]

public string FirstMidName { get; set; }

[Display(Name = "Full Name")]

public string FullName

{

get

{

return LastName + ", " + FirstMidName;

}

}

}

}

## Update Instructor and Student

In Instructor.cs, derive the Instructor class from the Person class and remove the key and name fields. The code will look like the following example:

C#Copy

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models

{

public class Instructor : Person

{

[DataType(DataType.Date)]

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

[Display(Name = "Hire Date")]

public DateTime HireDate { get; set; }

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

public OfficeAssignment OfficeAssignment { get; set; }

}

}

Make the same changes in Student.cs.

C#Copy

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models

{

public class Student : Person

{

[DataType(DataType.Date)]

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

[Display(Name = "Enrollment Date")]

public DateTime EnrollmentDate { get; set; }

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

}

}

## Add Person to the model

Add the Person entity type to SchoolContext.cs. The new lines are highlighted.

C#Copy

using ContosoUniversity.Models;

using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data

{

public class SchoolContext : DbContext

{

public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)

{

}

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

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

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

public DbSet<Department> Departments { get; set; }

public DbSet<Instructor> Instructors { get; set; }

public DbSet<OfficeAssignment> OfficeAssignments { get; set; }

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

public DbSet<Person> People { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Course>().ToTable("Course");

modelBuilder.Entity<Enrollment>().ToTable("Enrollment");

modelBuilder.Entity<Student>().ToTable("Student");

modelBuilder.Entity<Department>().ToTable("Department");

modelBuilder.Entity<Instructor>().ToTable("Instructor");

modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");

modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");

modelBuilder.Entity<Person>().ToTable("Person");

modelBuilder.Entity<CourseAssignment>()

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

}

}

}

This is all that the Entity Framework needs in order to configure table-per-hierarchy inheritance. As you'll see, when the database is updated, it will have a Person table in place of the Student and Instructor tables.

## Create and update migrations

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

.NET Core CLICopy

dotnet ef migrations add Inheritance

Don't run the database update command yet. That command will result in lost data because it will drop the Instructor table and rename the Student table to Person. You need to provide custom code to preserve existing data.

Open Migrations/<timestamp>\_Inheritance.cs and replace the Up method with the following code:

C#Copy

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.DropForeignKey(

name: "FK\_Enrollment\_Student\_StudentID",

table: "Enrollment");

migrationBuilder.DropIndex(name: "IX\_Enrollment\_StudentID", table: "Enrollment");

migrationBuilder.RenameTable(name: "Instructor", newName: "Person");

migrationBuilder.AddColumn<DateTime>(name: "EnrollmentDate", table: "Person", nullable: true);

migrationBuilder.AddColumn<string>(name: "Discriminator", table: "Person", nullable: false, maxLength: 128, defaultValue: "Instructor");

migrationBuilder.AlterColumn<DateTime>(name: "HireDate", table: "Person", nullable: true);

migrationBuilder.AddColumn<int>(name: "OldId", table: "Person", nullable: true);

// Copy existing Student data into new Person table.

migrationBuilder.Sql("INSERT INTO dbo.Person (LastName, FirstName, HireDate, EnrollmentDate, Discriminator, OldId) SELECT LastName, FirstName, null AS HireDate, EnrollmentDate, 'Student' AS Discriminator, ID AS OldId FROM dbo.Student");

// Fix up existing relationships to match new PK's.

migrationBuilder.Sql("UPDATE dbo.Enrollment SET StudentId = (SELECT ID FROM dbo.Person WHERE OldId = Enrollment.StudentId AND Discriminator = 'Student')");

// Remove temporary key

migrationBuilder.DropColumn(name: "OldID", table: "Person");

migrationBuilder.DropTable(

name: "Student");

migrationBuilder.CreateIndex(

name: "IX\_Enrollment\_StudentID",

table: "Enrollment",

column: "StudentID");

migrationBuilder.AddForeignKey(

name: "FK\_Enrollment\_Person\_StudentID",

table: "Enrollment",

column: "StudentID",

principalTable: "Person",

principalColumn: "ID",

onDelete: ReferentialAction.Cascade);

}

This code takes care of the following database update tasks:

* Removes foreign key constraints and indexes that point to the Student table.
* Renames the Instructor table as Person and makes changes needed for it to store Student data:
* Adds nullable EnrollmentDate for students.
* Adds Discriminator column to indicate whether a row is for a student or an instructor.
* Makes HireDate nullable since student rows won't have hire dates.
* Adds a temporary field that will be used to update foreign keys that point to students. When you copy students into the Person table they will get new primary key values.
* Copies data from the Student table into the Person table. This causes students to get assigned new primary key values.
* Fixes foreign key values that point to students.
* Re-creates foreign key constraints and indexes, now pointing them to the Person table.

(If you had used GUID instead of integer as the primary key type, the student primary key values wouldn't have to change, and several of these steps could have been omitted.)

Run the database update command:

.NET Core CLICopy

dotnet ef database update

(In a production system you would make corresponding changes to the Down method in case you ever had to use that to go back to the previous database version. For this tutorial you won't be using the Down method.)

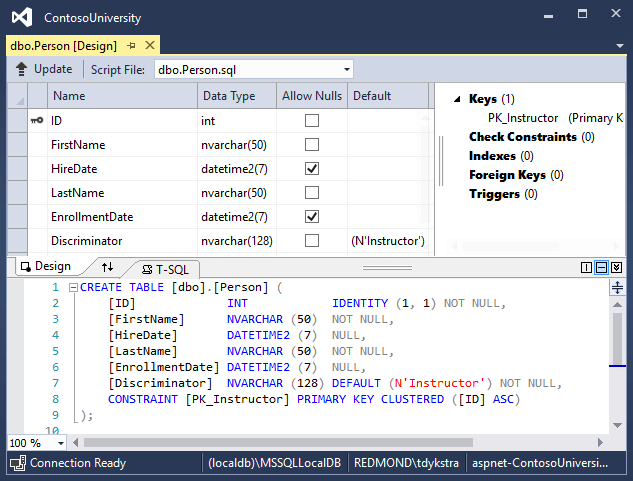
**Note**

It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors that you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there's no data to migrate, and the update-database command is more likely to complete without errors. To delete the database, use SSOX or run the database drop CLI command.

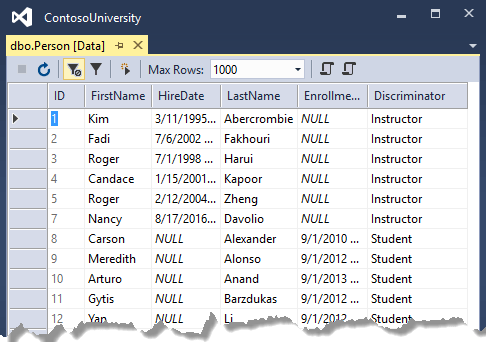
## Test the implementation

Run the app and try various pages. Everything works the same as it did before.

In **SQL Server Object Explorer**, expand **Data Connections/SchoolContext** and then **Tables**, and you see that the Student and Instructor tables have been replaced by a Person table. Open the Person table designer and you see that it has all of the columns that used to be in the Student and Instructor tables.



Right-click the Person table, and then click **Show Table Data** to see the discriminator column.



## Additional resources

For more information about inheritance in Entity Framework Core, see [Inheritance](https://docs.microsoft.com/en-us/ef/core/modeling/inheritance).

## Next steps

In this tutorial, you:

* Mapped inheritance to database
* Created the Person class
* Updated Instructor and Student
* Added Person to the model
* Created and update migrations
* Tested the implementation

Advance to the next tutorial to learn how to handle a variety of relatively advanced Entity Framework scenarios.

## Advanced Topics

In the previous tutorial, you implemented table-per-hierarchy inheritance. This tutorial introduces several topics that are useful to be aware of when you go beyond the basics of developing ASP.NET Core web applications that use Entity Framework Core.

In this tutorial, you:

* Perform raw SQL queries
* Call a query to return entities
* Call a query to return other types
* Call an update query
* Examine SQL queries
* Create an abstraction layer
* Learn about Automatic change detection
* Learn about EF Core source code and development plans
* Learn how to use dynamic LINQ to simplify code

## Prerequisites

* [Implement Inheritance](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/inheritance?view=aspnetcore-3.0)

## Perform raw SQL queries

One of the advantages of using the Entity Framework is that it avoids tying your code too closely to a particular method of storing data. It does this by generating SQL queries and commands for you, which also frees you from having to write them yourself. But there are exceptional scenarios when you need to run specific SQL queries that you have manually created. For these scenarios, the Entity Framework Code First API includes methods that enable you to pass SQL commands directly to the database. You have the following options in EF Core 1.0:

* Use the DbSet.FromSql method for queries that return entity types. The returned objects must be of the type expected by the DbSet object, and they're automatically tracked by the database context unless you [turn tracking off](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/crud?view=aspnetcore-3.0#no-tracking-queries).
* Use the Database.ExecuteSqlCommand for non-query commands.

If you need to run a query that returns types that aren't entities, you can use ADO.NET with the database connection provided by EF. The returned data isn't tracked by the database context, even if you use this method to retrieve entity types.

As is always true when you execute SQL commands in a web application, you must take precautions to protect your site against SQL injection attacks. One way to do that is to use parameterized queries to make sure that strings submitted by a web page can't be interpreted as SQL commands. In this tutorial you'll use parameterized queries when integrating user input into a query.

## Call a query to return entities

The DbSet<TEntity> class provides a method that you can use to execute a query that returns an entity of type TEntity. To see how this works you'll change the code in the Details method of the Department controller.

In DepartmentsController.cs, in the Details method, replace the code that retrieves a department with a FromSql method call, as shown in the following highlighted code:

C#Copy

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

{

if (id == null)

{

return NotFound();

}

string query = "SELECT \* FROM Department WHERE DepartmentID = {0}";

var department = await \_context.Departments

.FromSql(query, id)

.Include(d => d.Administrator)

.AsNoTracking()

.FirstOrDefaultAsync();

if (department == null)

{

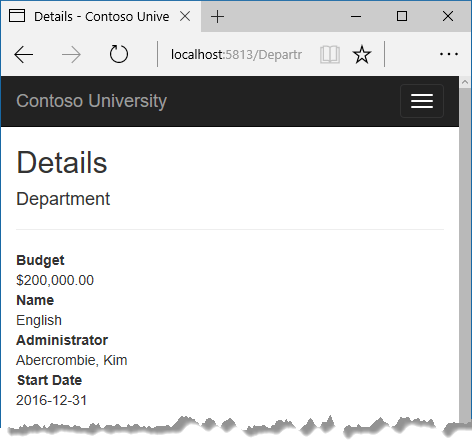
return NotFound();

}

return View(department);

}

To verify that the new code works correctly, select the **Departments** tab and then **Details** for one of the departments.



## Call a query to return other types

Earlier you created a student statistics grid for the About page that showed the number of students for each enrollment date. You got the data from the Students entity set (\_context.Students) and used LINQ to project the results into a list of EnrollmentDateGroup view model objects. Suppose you want to write the SQL itself rather than using LINQ. To do that you need to run a SQL query that returns something other than entity objects. In EF Core 1.0, one way to do that is write ADO.NET code and get the database connection from EF.

In HomeController.cs, replace the About method with the following code:

C#Copy

public async Task<ActionResult> About()

{

List<EnrollmentDateGroup> groups = new List<EnrollmentDateGroup>();

var conn = \_context.Database.GetDbConnection();

try

{

await conn.OpenAsync();

using (var command = conn.CreateCommand())

{

string query = "SELECT EnrollmentDate, COUNT(\*) AS StudentCount "

+ "FROM Person "

+ "WHERE Discriminator = 'Student' "

+ "GROUP BY EnrollmentDate";

command.CommandText = query;

DbDataReader reader = await command.ExecuteReaderAsync();

if (reader.HasRows)

{

while (await reader.ReadAsync())

{

var row = new EnrollmentDateGroup { EnrollmentDate = reader.GetDateTime(0), StudentCount = reader.GetInt32(1) };

groups.Add(row);

}

}

reader.Dispose();

}

}

finally

{

conn.Close();

}

return View(groups);

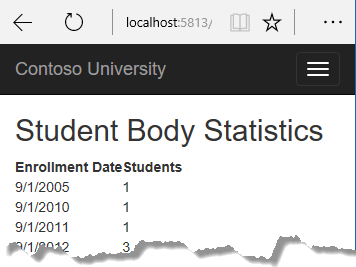
}

Add a using statement:

C#Copy

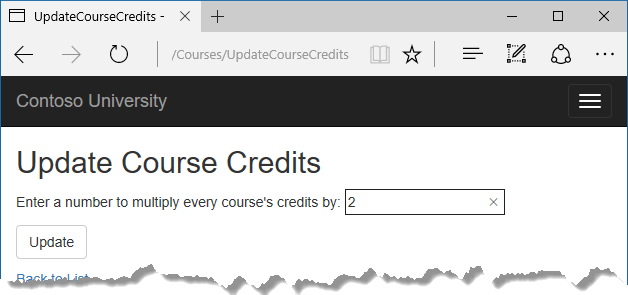
using System.Data.Common;

Run the app and go to the About page. It displays the same data it did before.



## Call an update query

Suppose Contoso University administrators want to perform global changes in the database, such as changing the number of credits for every course. If the university has a large number of courses, it would be inefficient to retrieve them all as entities and change them individually. In this section you'll implement a web page that enables the user to specify a factor by which to change the number of credits for all courses, and you'll make the change by executing a SQL UPDATE statement. The web page will look like the following illustration:



In CoursesController.cs, add UpdateCourseCredits methods for HttpGet and HttpPost:

C#Copy

public IActionResult UpdateCourseCredits()

{

return View();

}

C#Copy

[HttpPost]

public async Task<IActionResult> UpdateCourseCredits(int? multiplier)

{

if (multiplier != null)

{

ViewData["RowsAffected"] =

await \_context.Database.ExecuteSqlCommandAsync(

"UPDATE Course SET Credits = Credits \* {0}",

parameters: multiplier);

}

return View();

}

When the controller processes an HttpGet request, nothing is returned in ViewData["RowsAffected"], and the view displays an empty text box and a submit button, as shown in the preceding illustration.

When the **Update** button is clicked, the HttpPost method is called, and multiplier has the value entered in the text box. The code then executes the SQL that updates courses and returns the number of affected rows to the view in ViewData. When the view gets a RowsAffected value, it displays the number of rows updated.

In **Solution Explorer**, right-click the Views/Courses folder, and then click **Add > New Item**.

In the **Add New Item** dialog, click **ASP.NET Core** under **Installed** in the left pane, click **Razor View**, and name the new view UpdateCourseCredits.cshtml.

In Views/Courses/UpdateCourseCredits.cshtml, replace the template code with the following code:

HTMLCopy

@{

ViewBag.Title = "UpdateCourseCredits";

}

<h2>Update Course Credits</h2>

@if (ViewData["RowsAffected"] == null)

{

<form asp-action="UpdateCourseCredits">

<div class="form-actions no-color">

<p>

Enter a number to multiply every course's credits by: @Html.TextBox("multiplier")

</p>

<p>

<input type="submit" value="Update" class="btn btn-default" />

</p>

</div>

</form>

}

@if (ViewData["RowsAffected"] != null)

{

<p>

Number of rows updated: @ViewData["RowsAffected"]

</p>

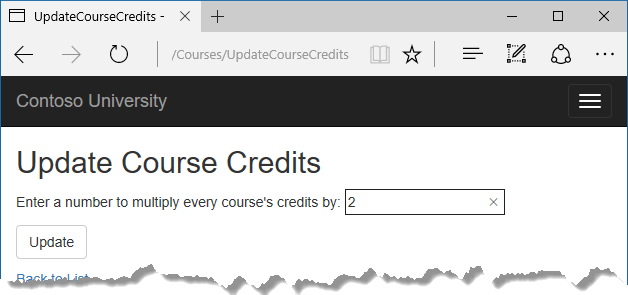
}

<div>

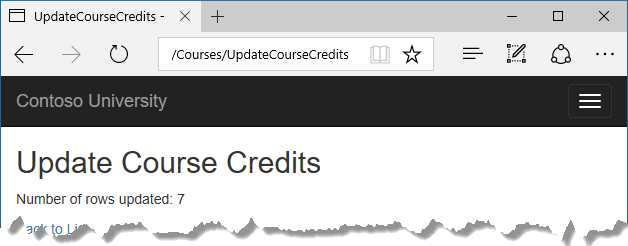
@Html.ActionLink("Back to List", "Index")

</div>

Run the UpdateCourseCredits method by selecting the **Courses** tab, then adding "/UpdateCourseCredits" to the end of the URL in the browser's address bar (for example: http://localhost:5813/Courses/UpdateCourseCredits). Enter a number in the text box:



Click **Update**. You see the number of rows affected:



Click **Back to List** to see the list of courses with the revised number of credits.

Note that production code would ensure that updates always result in valid data. The simplified code shown here could multiply the number of credits enough to result in numbers greater than 5. (The Credits property has a [Range(0, 5)] attribute.) The update query would work but the invalid data could cause unexpected results in other parts of the system that assume the number of credits is 5 or less.

For more information about raw SQL queries, see [Raw SQL Queries](https://docs.microsoft.com/en-us/ef/core/querying/raw-sql).

## Examine SQL queries

Sometimes it's helpful to be able to see the actual SQL queries that are sent to the database. Built-in logging functionality for ASP.NET Core is automatically used by EF Core to write logs that contain the SQL for queries and updates. In this section you'll see some examples of SQL logging.

Open StudentsController.cs and in the Details method set a breakpoint on the if (student == null) statement.

Run the app in debug mode, and go to the Details page for a student.

Go to the **Output** window showing debug output, and you see the query:

Copy

Microsoft.EntityFrameworkCore.Database.Command:Information: Executed DbCommand (56ms) [Parameters=[@\_\_id\_0='?'], CommandType='Text', CommandTimeout='30']

SELECT TOP(2) [s].[ID], [s].[Discriminator], [s].[FirstName], [s].[LastName], [s].[EnrollmentDate]

FROM [Person] AS [s]

WHERE ([s].[Discriminator] = N'Student') AND ([s].[ID] = @\_\_id\_0)

ORDER BY [s].[ID]

Microsoft.EntityFrameworkCore.Database.Command:Information: Executed DbCommand (122ms) [Parameters=[@\_\_id\_0='?'], CommandType='Text', CommandTimeout='30']

SELECT [s.Enrollments].[EnrollmentID], [s.Enrollments].[CourseID], [s.Enrollments].[Grade], [s.Enrollments].[StudentID], [e.Course].[CourseID], [e.Course].[Credits], [e.Course].[DepartmentID], [e.Course].[Title]

FROM [Enrollment] AS [s.Enrollments]

INNER JOIN [Course] AS [e.Course] ON [s.Enrollments].[CourseID] = [e.Course].[CourseID]

INNER JOIN (

SELECT TOP(1) [s0].[ID]

FROM [Person] AS [s0]

WHERE ([s0].[Discriminator] = N'Student') AND ([s0].[ID] = @\_\_id\_0)

ORDER BY [s0].[ID]

) AS [t] ON [s.Enrollments].[StudentID] = [t].[ID]

ORDER BY [t].[ID]

You'll notice something here that might surprise you: the SQL selects up to 2 rows (TOP(2)) from the Person table. The SingleOrDefaultAsync method doesn't resolve to 1 row on the server. Here's why:

* If the query would return multiple rows, the method returns null.
* To determine whether the query would return multiple rows, EF has to check if it returns at least 2.

Note that you don't have to use debug mode and stop at a breakpoint to get logging output in the **Output** window. It's just a convenient way to stop the logging at the point you want to look at the output. If you don't do that, logging continues and you have to scroll back to find the parts you're interested in.

## Create an abstraction layer

Many developers write code to implement the repository and unit of work patterns as a wrapper around code that works with the Entity Framework. These patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD). However, writing additional code to implement these patterns isn't always the best choice for applications that use EF, for several reasons:

* The EF context class itself insulates your code from data-store-specific code.
* The EF context class can act as a unit-of-work class for database updates that you do using EF.
* EF includes features for implementing TDD without writing repository code.

For information about how to implement the repository and unit of work patterns, see [the Entity Framework 5 version of this tutorial series](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application).

Entity Framework Core implements an in-memory database provider that can be used for testing. For more information, see [Test with InMemory](https://docs.microsoft.com/en-us/ef/core/miscellaneous/testing/in-memory).

## Automatic change detection

The Entity Framework determines how an entity has changed (and therefore which updates need to be sent to the database) by comparing the current values of an entity with the original values. The original values are stored when the entity is queried or attached. Some of the methods that cause automatic change detection are the following:

* DbContext.SaveChanges
* DbContext.Entry
* ChangeTracker.Entries

If you're tracking a large number of entities and you call one of these methods many times in a loop, you might get significant performance improvements by temporarily turning off automatic change detection using the ChangeTracker.AutoDetectChangesEnabled property. For example:

C#Copy

\_context.ChangeTracker.AutoDetectChangesEnabled = false;

## EF Core source code and development plans

The Entity Framework Core source is at <https://github.com/aspnet/EntityFrameworkCore>. The EF Core repository contains nightly builds, issue tracking, feature specs, design meeting notes, and [the roadmap for future development](https://github.com/aspnet/EntityFrameworkCore/wiki/Roadmap). You can file or find bugs, and contribute.

Although the source code is open, Entity Framework Core is fully supported as a Microsoft product. The Microsoft Entity Framework team keeps control over which contributions are accepted and tests all code changes to ensure the quality of each release.

## Reverse engineer from existing database

To reverse engineer a data model including entity classes from an existing database, use the [scaffold-dbcontext](https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/powershell#scaffold-dbcontext) command. See the [getting-started tutorial](https://docs.microsoft.com/en-us/ef/core/get-started/aspnetcore/existing-db).

## Use dynamic LINQ to simplify code

The [third tutorial in this series](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/sort-filter-page?view=aspnetcore-3.0) shows how to write LINQ code by hard-coding column names in a switch statement. With two columns to choose from, this works fine, but if you have many columns the code could get verbose. To solve that problem, you can use the EF.Property method to specify the name of the property as a string. To try out this approach, replace the Index method in the StudentsController with the following code.

C#Copy

public async Task<IActionResult> Index(

string sortOrder,

string currentFilter,

string searchString,

int? pageNumber)

{

ViewData["CurrentSort"] = sortOrder;

ViewData["NameSortParm"] =

String.IsNullOrEmpty(sortOrder) ? "LastName\_desc" : "";

ViewData["DateSortParm"] =

sortOrder == "EnrollmentDate" ? "EnrollmentDate\_desc" : "EnrollmentDate";

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));

}

if (string.IsNullOrEmpty(sortOrder))

{

sortOrder = "LastName";

}

bool descending = false;

if (sortOrder.EndsWith("\_desc"))

{

sortOrder = sortOrder.Substring(0, sortOrder.Length - 5);

descending = true;

}

if (descending)

{

students = students.OrderByDescending(e => EF.Property<object>(e, sortOrder));

}

else

{

students = students.OrderBy(e => EF.Property<object>(e, sortOrder));

}

int pageSize = 3;

return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(),

pageNumber ?? 1, pageSize));

}

## Acknowledgments

Tom Dykstra and Rick Anderson (twitter @RickAndMSFT) wrote this tutorial. Rowan Miller, Diego Vega, and other members of the Entity Framework team assisted with code reviews and helped debug issues that arose while we were writing code for the tutorials. John Parente and Paul Goldman worked on updating the tutorial for ASP.NET Core 2.2.

## Troubleshoot common errors

### ContosoUniversity.dll used by another process

Error message:

Cannot open '...bin\Debug\netcoreapp1.0\ContosoUniversity.dll' for writing -- 'The process cannot access the file '...\bin\Debug\netcoreapp1.0\ContosoUniversity.dll' because it is being used by another process.

Solution:

Stop the site in IIS Express. Go to the Windows System Tray, find IIS Express and right-click its icon, select the Contoso University site, and then click **Stop Site**.

### Migration scaffolded with no code in Up and Down methods

Possible cause:

The EF CLI commands don't automatically close and save code files. If you have unsaved changes when you run the migrations add command, EF won't find your changes.

Solution:

Run the migrations remove command, save your code changes and rerun the migrations add command.

### Errors while running database update

It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there's no data to migrate, and the update-database command is much more likely to complete without errors.

The simplest approach is to rename the database in appsettings.json. The next time you run database update, a new database will be created.

To delete a database in SSOX, right-click the database, click **Delete**, and then in the **Delete Database** dialog box select **Close existing connections** and click **OK**.

To delete a database by using the CLI, run the database drop CLI command:

.NET Core CLICopy

dotnet ef database drop

### Error locating SQL Server instance

Error Message:

A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: SQL Network Interfaces, error: 26 - Error Locating Server/Instance Specified)

Solution:

Check the connection string. If you have manually deleted the database file, change the name of the database in the construction string to start over with a new database.

## Additional resources

For more information about EF Core, see the [Entity Framework Core documentation](https://docs.microsoft.com/en-us/ef/core). A book is also available: [Entity Framework Core in Action](https://www.manning.com/books/entity-framework-core-in-action).

For information on how to deploy a web app, see [Host and deploy ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/index?view=aspnetcore-3.0).

For information about other topics related to ASP.NET Core MVC, such as authentication and authorization, see [Introduction to ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/index?view=aspnetcore-3.0).

## Next steps

In this tutorial, you:

* Performed raw SQL queries
* Called a query to return entities
* Called a query to return other types
* Called an update query
* Examined SQL queries
* Created an abstraction layer
* Learned about Automatic change detection
* Learned about EF Core source code and development plans
* Learned how to use dynamic LINQ to simplify code

This completes this series of tutorials on using the Entity Framework Core in an ASP.NET Core MVC application. This series worked with a new database; an alternative is to reverse engineer a model from an existing database.

## Get the code

[Download or view the completed application.](https://github.com/aspnet/AspNetCore.Docs/tree/master/aspnetcore/data/ef-mvc/intro/samples/cu-final)