

```

    {
        table.PrimaryKey("PK_Department", x => x.DepartmentID);
        table.ForeignKey(
            name: "FK_Department_Instructor_InstructorID",
            column: x => x.InstructorID,
            principalTable: "Instructor",
            principalColumn: "ID",
            onDelete: ReferentialAction.Restrict);
    });

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

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

```

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

Save your changes and build the project.

## Change the connection string

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

JSON

```

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

```

Save your change to `appsettings.json`.

### ⓘ Note

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

```
.NET CLI
```

```
dotnet ef database drop
```

## Update the database

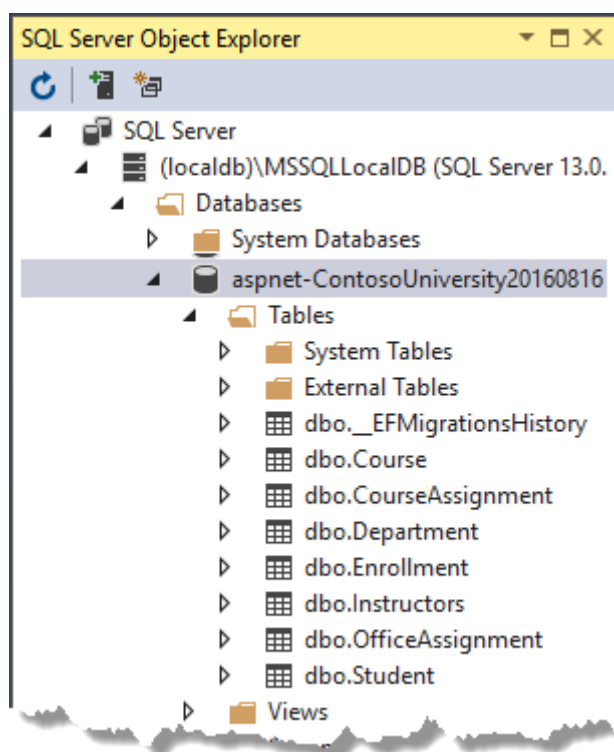
After you have changed the database name or deleted the database, run the `database update` command in the command window to execute the migrations.

```
.NET CLI
```

```
dotnet ef database update
```

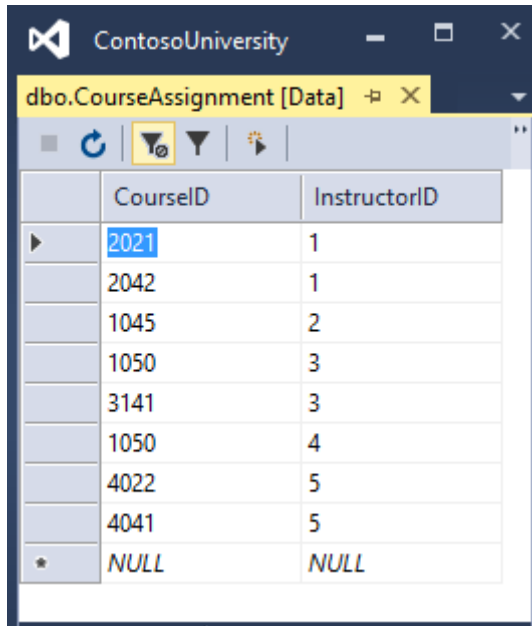
Run the app to cause the `DbInitializer.Initialize` method to run and populate the new database.

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



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

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



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

## Get the code

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

## Next steps

In this tutorial, you:

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

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

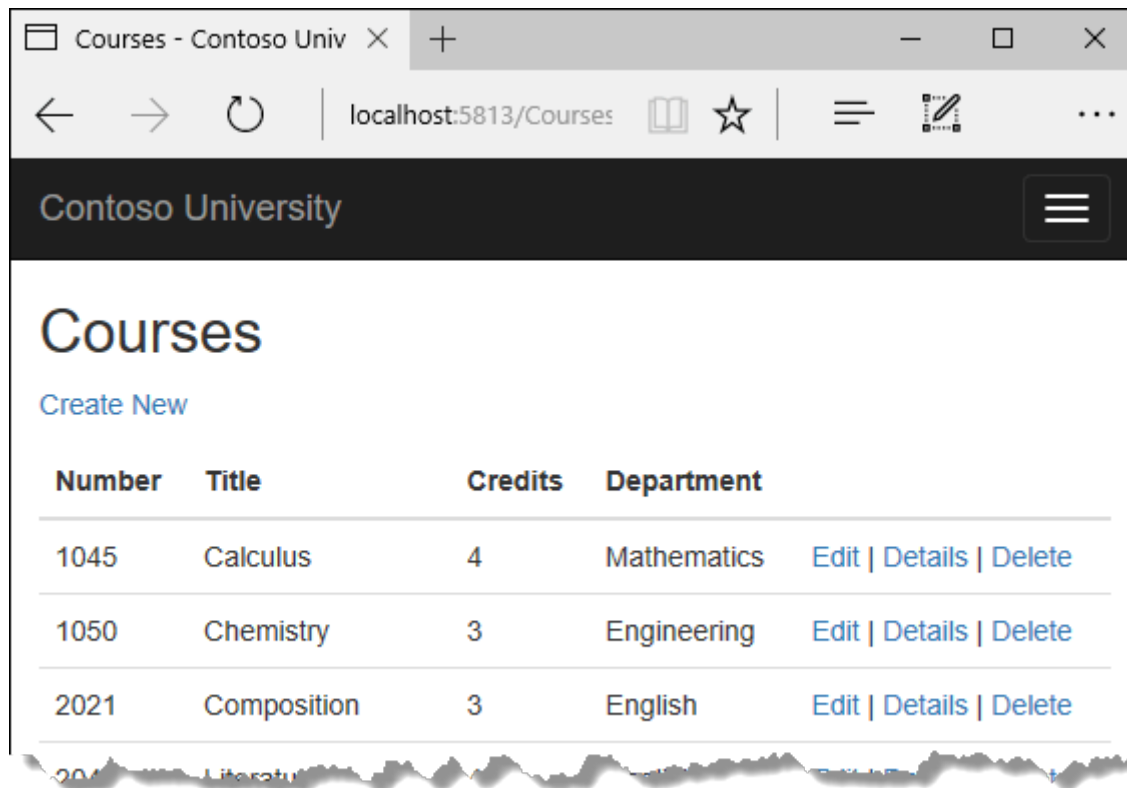
Next: Access related data

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

Article • 04/10/2024

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

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



Instructors - Contoso UI X +

localhost:5813/Instructors/Index/1?

## Contoso University

# Instructors

[Create New](#)

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

## Courses Taught by Selected Instructor

	Number	Title	Department
<a href="#">Select</a>	2021	Composition	English
<a href="#">Select</a>	2042	Literature	English

## Students Enrolled in Selected Course

Name	Grade
Alonso, Meredith	B
Li, Yan	B

In this tutorial, you:

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

# Prerequisites

- [Create a complex data model](#)

## Learn how to load related data

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

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

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

Query: all Department entities and related Course entities

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

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

Query: all Department rows

Query: Course rows related to Department d

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

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

Query: all Department rows

Query: Course rows related to Department d

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

## Performance considerations

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

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

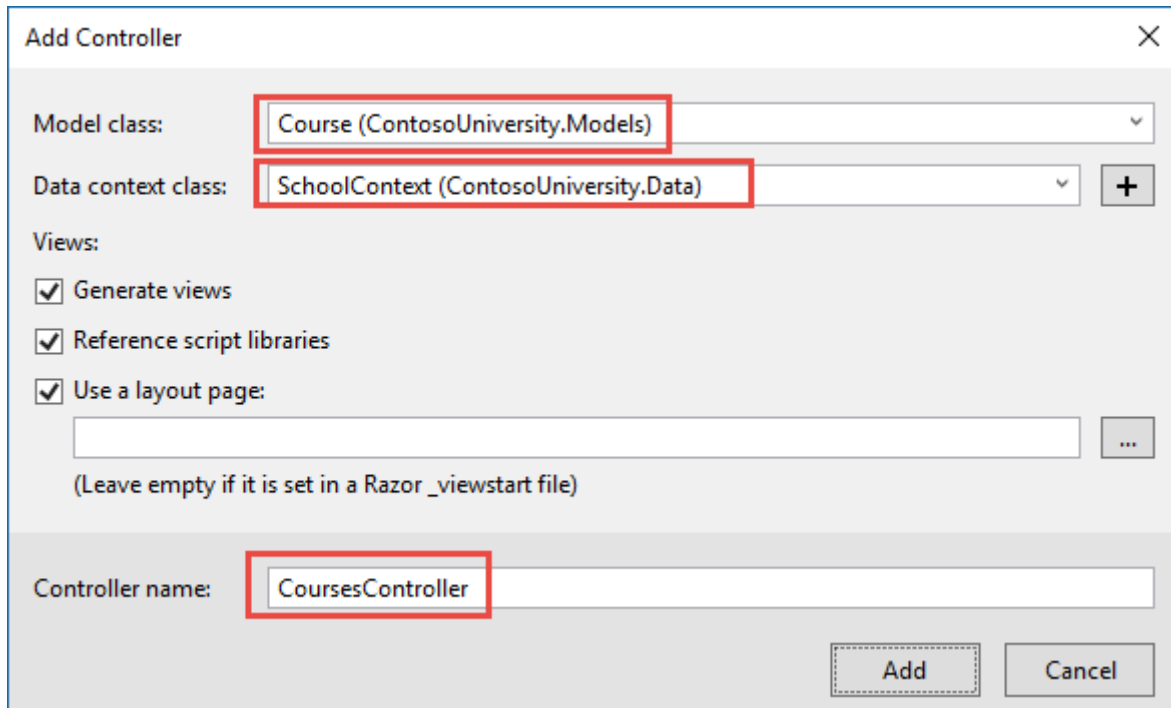
## Create a Courses page

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

Create a controller named `CoursesController` for the `Course` entity type, using the same options for the **MVC Controller with views, using Entity Framework** scaffolder that you



did earlier for the `StudentsController`, as shown in the following illustration:



The screenshot shows the 'Add Controller' dialog box. The 'Model class' dropdown is set to 'Course (ContosoUniversity.Models)'. The 'Data context class' dropdown is set to 'SchoolContext (ContosoUniversity.Data)'. The 'Views' section has three checkboxes: 'Generate views' (checked), 'Reference script libraries' (checked), and 'Use a layout page' (checked). The 'Controller name' text box contains 'CoursesController'. The 'Add' button is highlighted.

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#

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

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

```
CSHTML

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

```

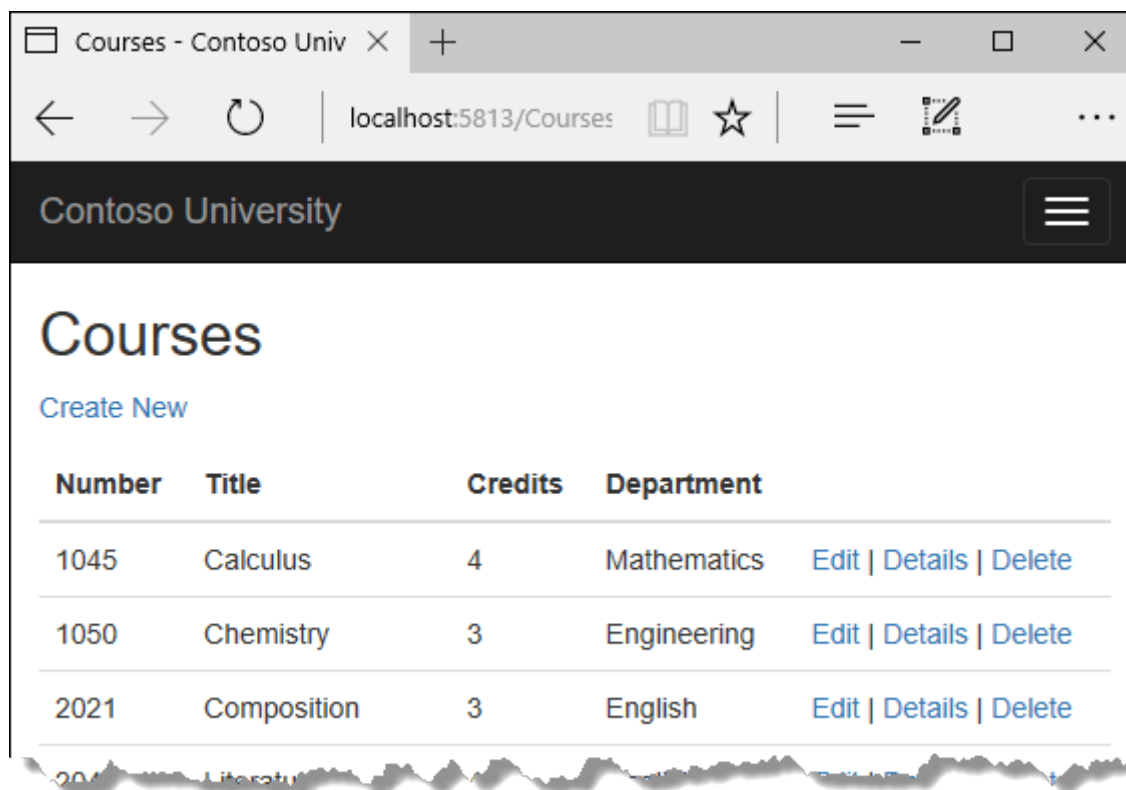
You've made the following changes to the scaffolded code:

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

HTML

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

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



## 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:

Instructors - Contoso UI X + - □ X

localhost:5813/Instructors/Index/1? ☆ | ≡ | ...

Contoso University

# Instructors

Create New

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

## Courses Taught by Selected Instructor

	Number	Title	Department
Select	2021	Composition	English
Select	2042	Literature	English

## Students Enrolled in Selected Course

Name	Grade
Alonso, Meredith	B
Li, Yan	B

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

- The list of instructors displays related data from the `OfficeAssignment` entity. The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. You'll use eager loading for the `OfficeAssignment` entities. As explained earlier, eager loading is typically more efficient when you need the

related data for all retrieved rows of the primary table. In this case, you want to display office assignments for all displayed instructors.

- When the user selects an instructor, related `Course` entities are displayed. The `Instructor` and `Course` entities are in a many-to-many relationship. You'll use eager loading for the `Course` entities and their related `Department` entities. In this case, separate queries might be more efficient because you need courses only for the selected instructor. However, this example shows how to use eager loading for navigation properties within entities that are themselves in navigation properties.
- When the user selects a course, related data from the `Enrollments` entity set is displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship. You'll use separate queries for `Enrollment` entities and their related `Student` entities.

## Create a view model for the Instructor Index view

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

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

C#

```
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:

**Add Controller**

Model class: **Instructor (ContosoUniversity.Models)**

Data context class: **SchoolContext (ContosoUniversity.Data)** **+**

Views:

- ☒ Generate views
- ☒ Reference script libraries
- ☒ Use a layout page:  **...**

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

Controller name: **InstructorsController**

**Add** **Cancel**

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

```
C#

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#

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#

```

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`.

You can read more about including multiple levels of related data [here](#).

C#

```
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#

```
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#



```

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#

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

Instead of:

C#

```
.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#

```

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

```

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

## Tracking vs no-tracking

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

In some cases a tracking query is more efficient than a no-tracking query. For more information, see [Tracking vs. No-Tracking Queries](#).

## Modify the Instructor Index view

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

CSHTML

```
@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 = "table-success";
}
<tr class="@selectedRow">
    <td>
        @Html.DisplayFor(modelItem => item.LastName)
    </td>
    <td>
        @Html.DisplayFor(modelItem => item.FirstMidName)
    </td>
    <td>
        @Html.DisplayFor(modelItem => item.HireDate)
    </td>
    <td>
        @if (item.OfficeAssignment != null)
        {
            @item.OfficeAssignment.Location
        }
    </td>
    <td>
        @foreach (var course in item.CourseAssignments)
        {
            @course.Course.CourseID @course.Course.Title <br />
        }
    </td>
    <td>
        <a asp-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.)

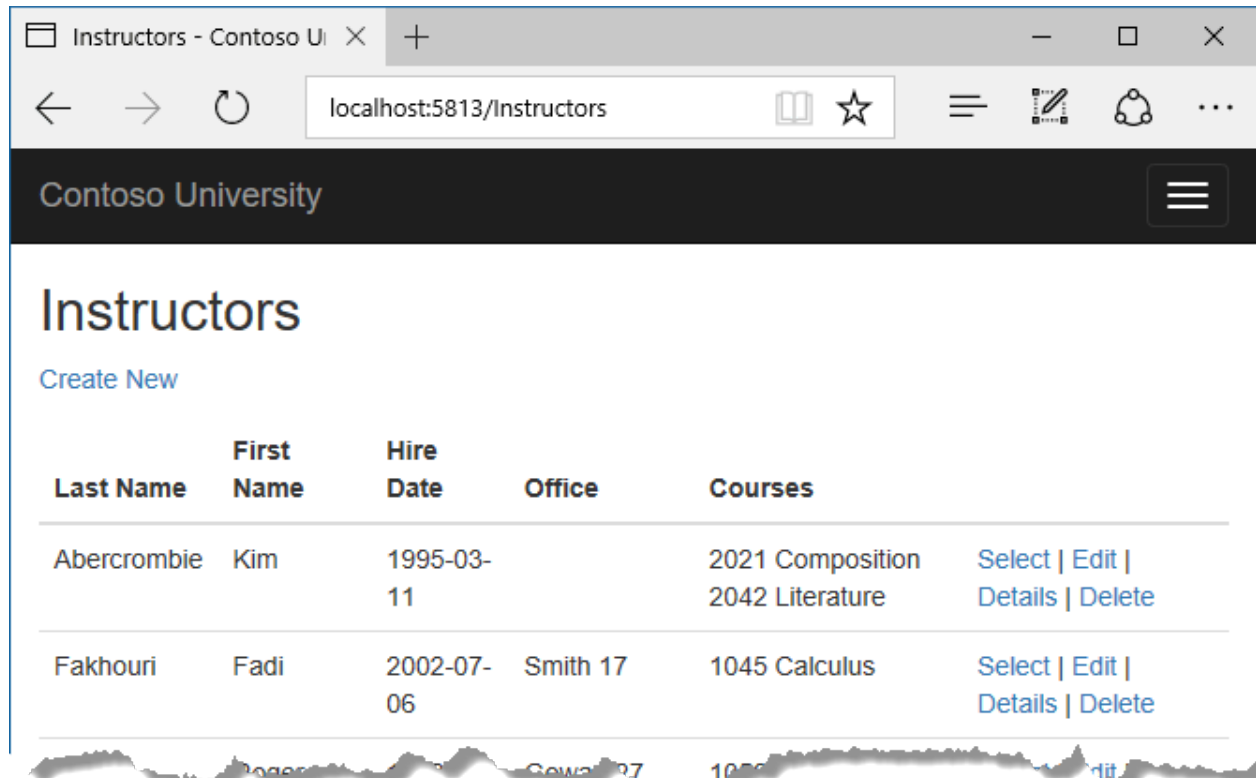
```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

- Added a **Courses** column that displays courses taught by each instructor. For more information, see the [Explicit line transition](#) section of the Razor syntax article.
- Added code that conditionally adds a Bootstrap CSS class to the `tr` element of the selected instructor. This class sets a background color for the selected row.
- 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.

CSSHTML

```
<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.

```

@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.

The screenshot shows a web browser window with the address bar displaying 'localhost:5813/Instructors/Index/1'. The page has a dark header with 'Contoso University' and a hamburger menu icon. Below the header, the main section is titled 'Instructors' and includes a 'Create New' link. A table lists two instructors: Kim Abercrombie and Fadi Fakhouri. Kim Abercrombie is highlighted in green and is associated with courses 2021 Composition and 2042 Literature. Fadi Fakhouri is associated with course 1045 Calculus. Each instructor row has links for 'Select', 'Edit', 'Details', and 'Delete'. Below the instructors table, there is a section titled 'Courses Taught by Selected Instructor'. This section contains a table with three rows, each representing a course taught by the selected instructor (Kim Abercrombie). Each row has a 'Select' link, a course number, a title, and a department.

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

	Number	Title	Department
Select	2021	Composition	English
Select	2042	Literature	English

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

CSSHTML

```
@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>
            </tr>
        }
    </table>
}
```

```
        </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.

Contoso University

## Instructors

[Create New](#)

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

## Courses Taught by Selected Instructor

	Number	Title	Department
<a href="#">Select</a>	2021	Composition	English
<a href="#">Select</a>	2042	Literature	English

## Students Enrolled in Selected Course

Name	Grade
Alonso, Meredith	B
Li, Yan	B

## 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#

```
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.

## Get the code

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

## 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.

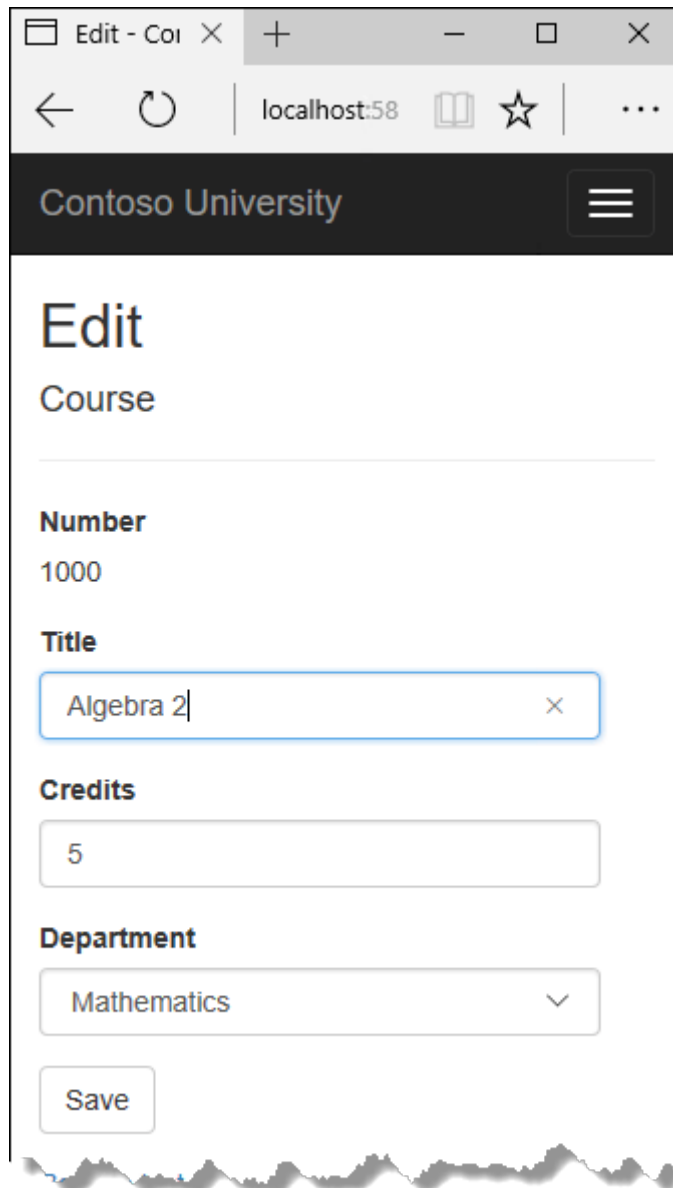
[Update related data](#)

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

Article • 04/10/2024

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.



The screenshot shows a web browser window with the address bar displaying 'localhost:58'. The page title is 'Contoso University'. The main content area is titled 'Edit Course'. Below the title, there are four form fields: 'Number' with the value '1000', 'Title' with the value 'Algebra 2', 'Credits' with the value '5', and 'Department' with the value 'Mathematics'. A 'Save' button is located at the bottom of the form.

Edit - Contoso Universit × + − □ ×

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

Contoso University ≡

## Edit

### Instructor

---

**Last Name**

**First Name**

**Hire Date**

**Office Location**

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

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](#)

# 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#

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

C#

```
[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#

```
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#

```

[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#

```
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#

```
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#

```
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#

```

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#

```

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.

CSHTML

```

<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" />
</div>

```

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.

CSHTML

```

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

```

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

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

```

@model ContosoUniversity.Models.Course

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

<h2>Delete</h2>

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

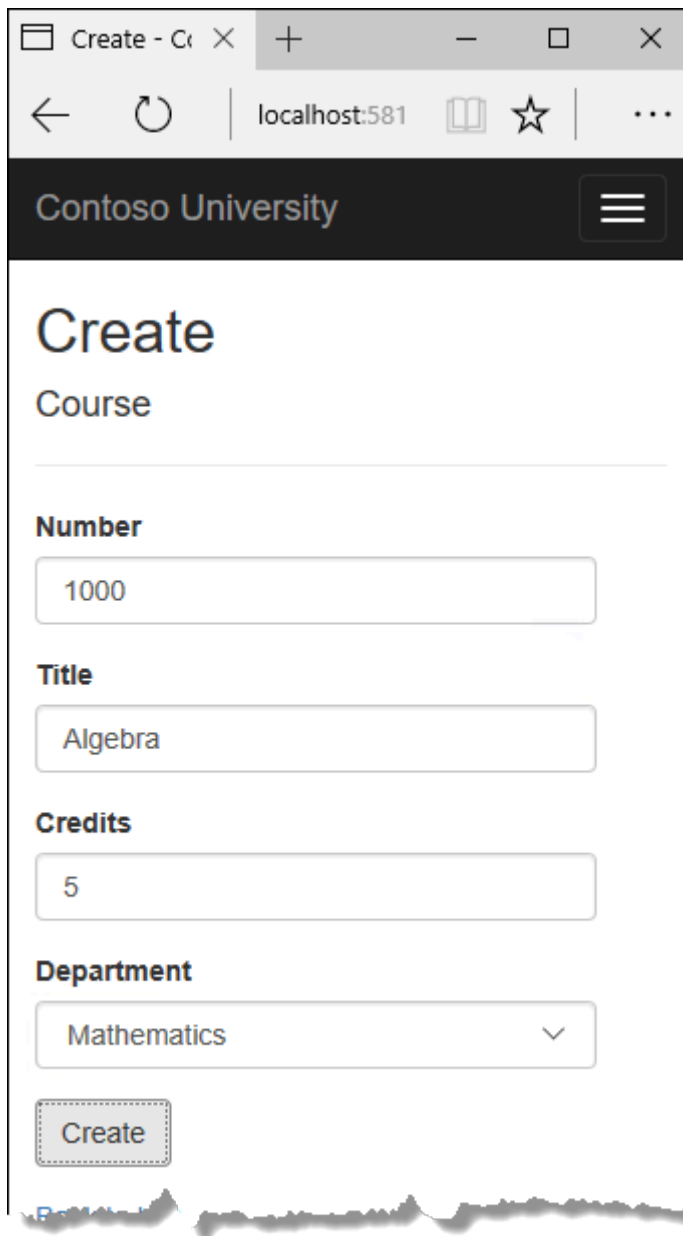
    <form asp-action="Delete">
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>

```

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:



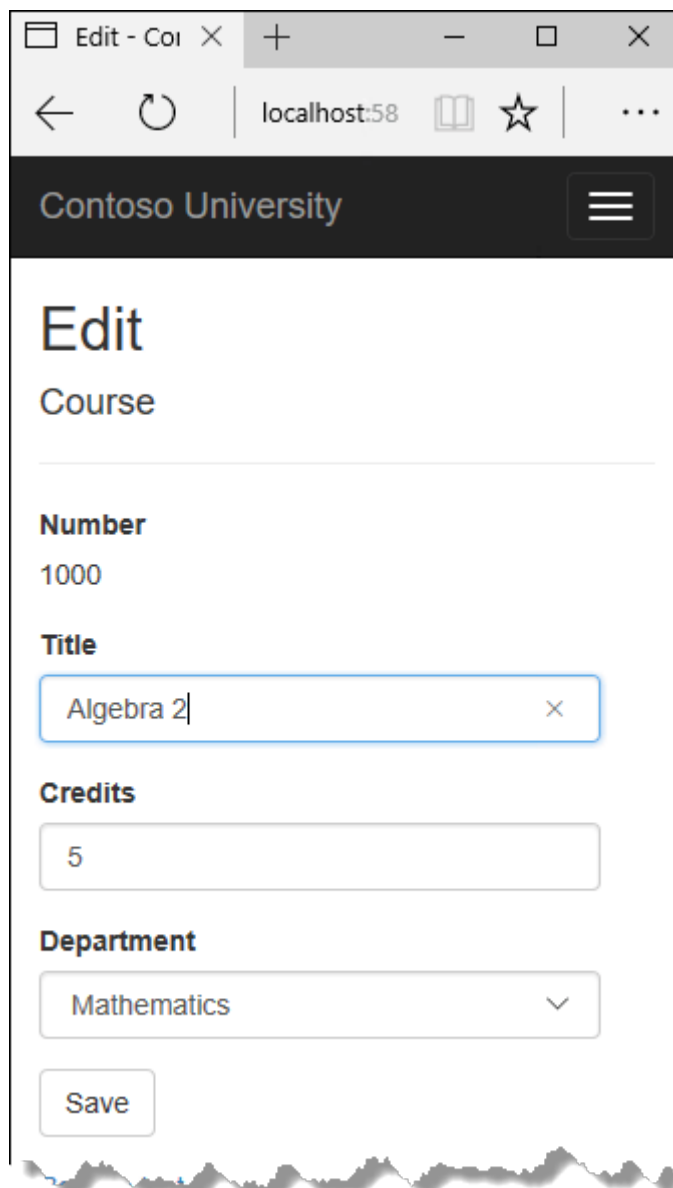
The screenshot shows a web browser window with the address bar displaying 'localhost:581'. The page title is 'Create - C'. The main content area is titled 'Create Course' and contains a form with the following fields:

- Number**: A text input field containing the value '1000'.
- Title**: A text input field containing the value 'Algebra'.
- Credits**: A text input field containing the value '5'.
- Department**: A dropdown menu with 'Mathematics' selected.
- Create**: A button at the bottom of the form.

The browser window also shows a navigation menu in the top right corner with three horizontal lines.

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.



Contoso University

## Edit Course

**Number**  
1000

**Title**  
Algebra 2

**Credits**  
5

**Department**  
Mathematics

Save

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#

```
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#

```
[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 declare the properties you want to include. This prevents over-posting, as explained in the [second tutorial](#).

C#

```

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#

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

- Saves the changes to the database.

## Update the Instructor Edit view

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

CSHTML

```
<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**.

Contoso University

## Edit Instructor

**Last Name**

Abercrombie

**First Name**

Kim

**Hire Date**

3/11/1995

**Office Location**

44/3P

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 checkboxes, as shown in the following screen shot:



Edit - Contoso Universit × + — □ ×

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

Contoso University ≡

## Edit

### Instructor

---

**Last Name**

**First Name**

**Hire Date**

**Office Location**

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

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 checkboxes. A checkbox 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 checkboxes 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 checkboxes, you'll use a view model class.

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

C#

```
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#

```
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 checkbox 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 checkboxes 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#

```

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

```

C#

```

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 overload that requires explicit approval and `CourseAssignments` isn't in the include list.

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

C#

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

in the navigation property.

C#

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

C#

```
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 checkboxes 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.

CSSHTML

```
<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>
  </div>
```

This code creates an HTML table that has three columns. In each column is a checkbox followed by a caption that consists of the course number and title. The checkboxes 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 checkbox 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 checkboxes which are selected.

When the checkboxes 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.

Edit - Contoso Universit × + − □ ×

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

Contoso University ≡

## Edit

### Instructor

---

**Last Name**

**First Name**

**Hire Date**

**Office Location**

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

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

#### ⓘ Note

The approach taken here to edit instructor course data works well when there'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#

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

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

    _context.Instructors.Remove(instructor);

    await _context.SaveChangesAsync();
    return RedirectToAction(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#

```
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#

```
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#

```
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 checkboxes 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](#).

CSHTML

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

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;

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

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
```

```

        {
            @:</tr><tr>
        }
        @:<td>
            <input type="checkbox"
                name="selectedCourses"
                value="@course.CourseID"
                @(Html.Raw(course.Assigned ?
"checked=\"checked\" : "")) />
            @course.CourseID @: @course.Title
        @:</td>
    }
    @:</tr>
}
</table>
</div>
</div>

```

Test by running the app and creating an instructor.

## Handling Transactions

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

## Get the code

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

## 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](#)

# Tutorial: Handle concurrency - ASP.NET MVC with EF Core

Article • 07/09/2024

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.

Departmen × Edit - Cont × + − □ ×

← → ↺

localhost:5813/Departr

📖

☆

⋮

Contoso University

☰

Edit

Department

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

Name

English

Budget

200000.00

Current value: \$50,000.00

Start Date

9/1/2007

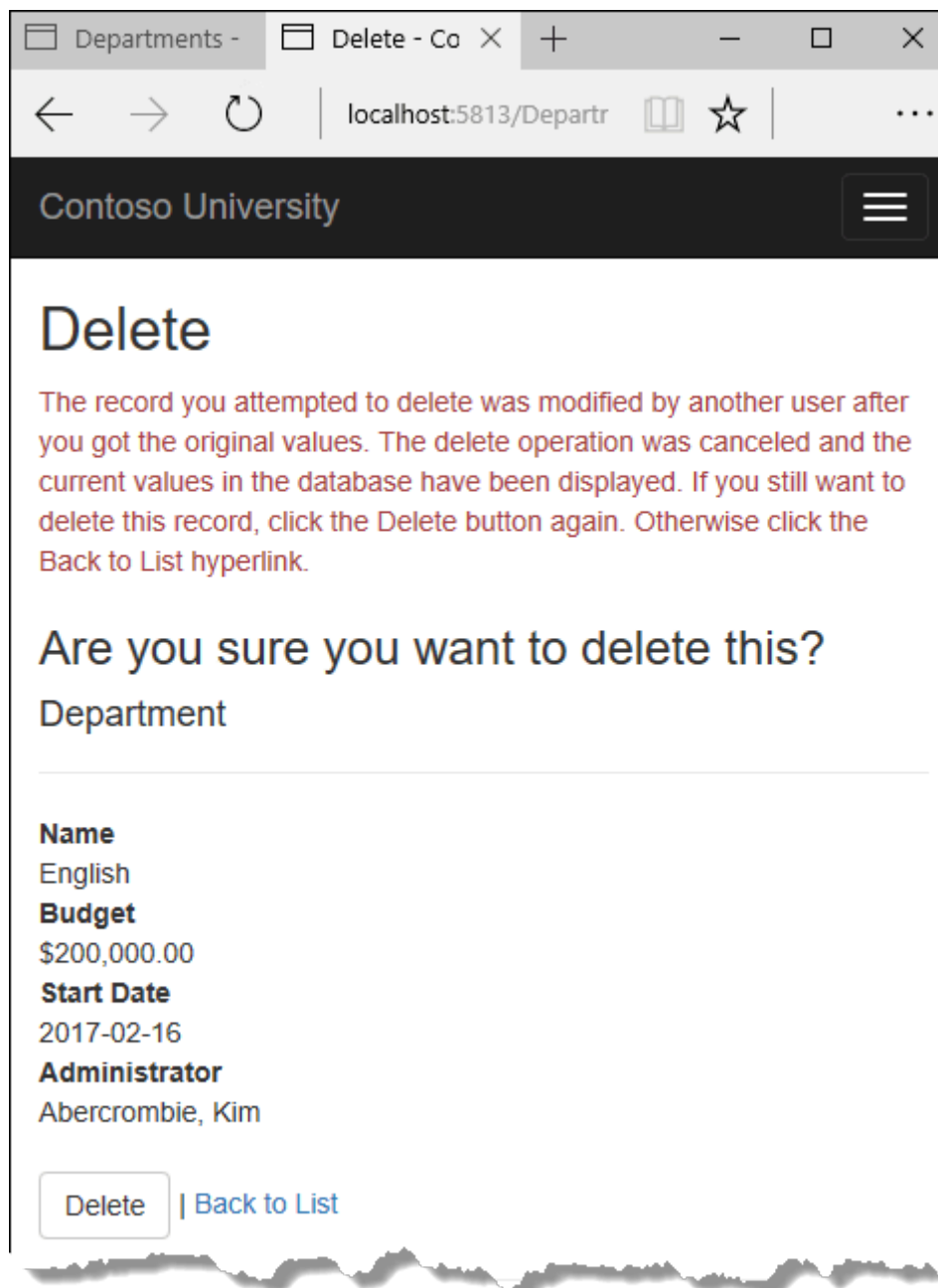
InstructorID

Abercrombie, Kim

▼

Save





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](#)

## Concurrency conflicts

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

### Pessimistic concurrency (locking)

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

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

### Optimistic Concurrency

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

Edit - Cont × + − □ ×

← ↻ | localhost:581 | ☆ | ...

Contoso University ☰

# Edit

## Department

---

**Budget**

**Administrator**

**Name**

**Start Date**

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

Edit - Cont ✕ + - □ ×

← ↻ | localhost:581 | 📖 ☆ | ...

Contoso University ☰

## Edit

Department

---

**Budget**

**Administrator**

**Name**

**Start Date**

Save

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

Departments - Contoso ✕ + - □ ×

← → ↻ | localhost:5813/Departments | 📖 ☆ | ☰ ✎ 👤 | ...

Contoso University ☰

## Departments

[Create New](#)

Name	Budget	Administrator	Start Date	
English	\$0.00	Abercrombie, Kim	2007-09-01	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Mathematics	\$100000.00	Eakheriri, Fadi	2007-09-01	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

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#

```
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 `IsRowVersion()` method (in `Data/SchoolContext.cs`) to specify the tracking property, as shown in the following example:

C#

```
modelBuilder.Entity<Department>()
    .Property(p => p.RowVersion).IsRowVersion();
```

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 CLI
```

```
dotnet ef migrations add RowVersion
```

```
.NET CLI
```

```
dotnet ef database update
```

## Create Departments controller and views

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

The screenshot shows the 'Add Controller' dialog box. The 'Model class' dropdown is set to 'Department (ContosoUniversity.Models)'. The 'Data context class' dropdown is set to 'SchoolContext (ContosoUniversity.Data)'. The 'Views' section has three checkboxes: 'Generate views' (checked), 'Reference script libraries' (checked), and 'Use a layout page' (checked). The 'Controller name' text box contains 'DepartmentsController'. The 'Add' button is highlighted with a dashed border.

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

```
C#
```

```
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.

CSHTML

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

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

<h2>Departments</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Administrator)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
```

```

        @Html.DisplayFor(modelItem =>
item.Administrator.FullName)
    </td>
    <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#

```

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#

```

[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#

```

_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#

```
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#

```
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#

```
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#

```
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.

CSHTML

```
@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**.

Contoso University

## Edit Department

**Name**

English

**Budget**

50000.00

**Start Date**

9/1/2007

**InstructorID**

Abercrombie, Kim

Save

The browser shows the Index page with the changed value.

Change a field in the second browser tab.



Departments - Edit - Conto × + - □ ×

localhost:5813/Departr ☆ ⋮

Contoso University ☰

## Edit

### Department

---

**Name**

English

**Budget**

200000.00 ×

**Start Date**

9/1/2007

**InstructorID**

Abercrombie, Kim ▼

Save

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

Department x Edit - Conto x + - □ ×

localhost:5813/Departr ☆ ...

## Contoso University

# Edit

## Department

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

**Name**

**Budget**

Current value: \$50,000.00

**Start Date**

**InstructorID**

Save

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

## Update the Delete page

For the Delete page, the Entity Framework detects concurrency conflicts caused by someone else editing the department in a similar manner. When the `HttpGet Delete` method displays the confirmation view, the view includes the original `RowVersion` value in a hidden field. That value is then available to the `HttpPost Delete` method that's called when the user confirms the deletion. When the Entity Framework creates the SQL

DELETE command, it includes a WHERE clause with the original `RowVersion` value. If the command results in zero rows affected (meaning the row was changed after the Delete confirmation page was displayed), a concurrency exception is thrown, and the `HttpGet Delete` method is called with an error flag set to true in order to redisplay the confirmation page with an error message. It's also possible that zero rows were affected because the row was deleted by another user, so in that case no error message is displayed.

## Update the Delete methods in the Departments controller

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

C#

```
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#  
  
[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#  
  
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#
```

```
public async Task<IActionResult> Delete(Department department)
```

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

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

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

## Update the Delete view

In `Views/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.

CSHTML

```
@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**:

Edit - Conto × Delete - Conto: + - □ ×

← → ↻ | localhost:5813/Departr | ☆ | ...

Contoso University

## Edit Department

**Name**

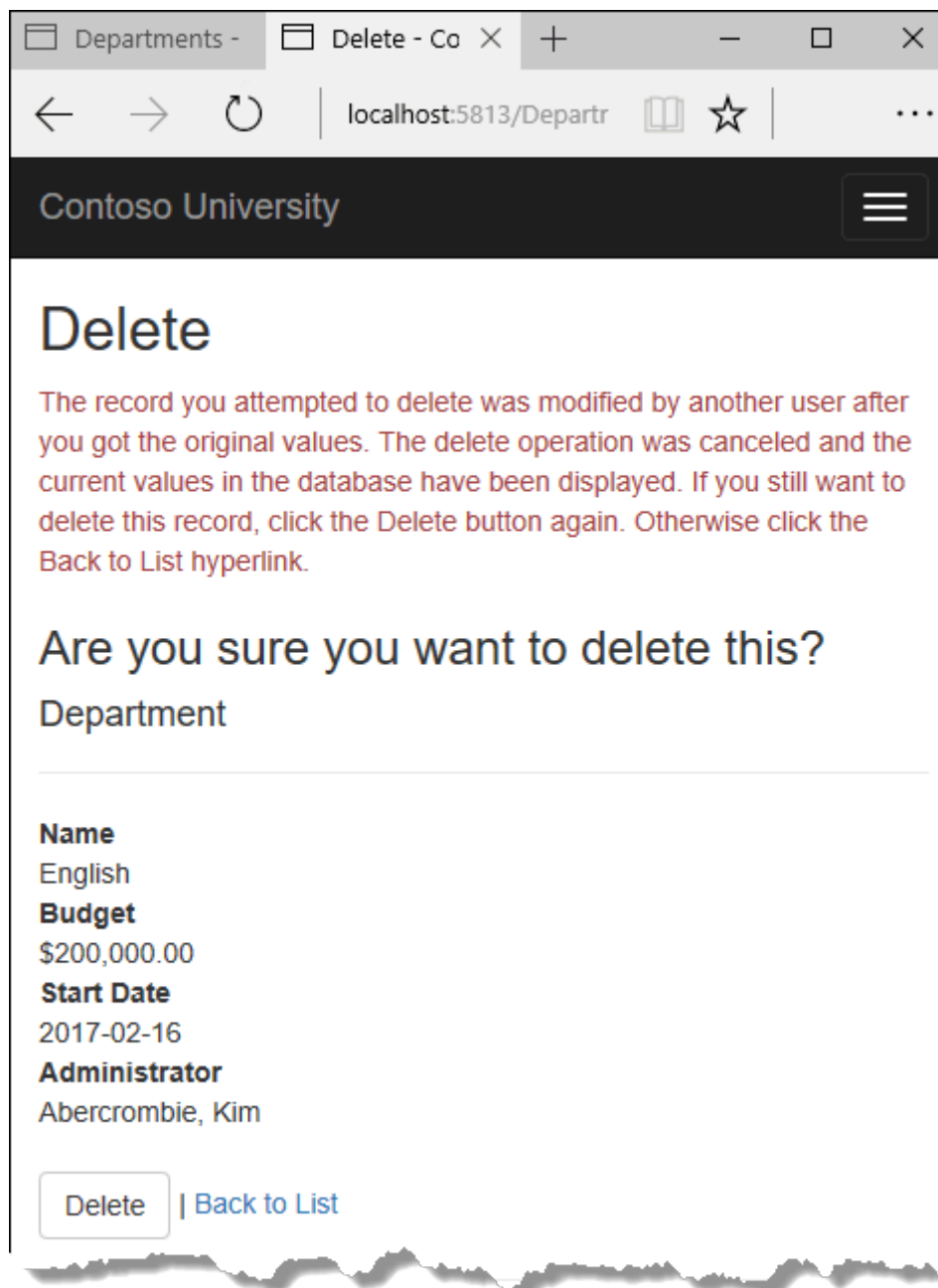
**Budget**

**Start Date**

**InstructorID**

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.

CSHTML

```
@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.

CSHTML

```

@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");}
    }

```

## Get the code

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

# Additional resources

For more information about how to handle concurrency in EF Core, see [Concurrency conflicts](#).

## 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.

**Next: Implement table-per-hierarchy inheritance**

# Tutorial: Implement inheritance - ASP.NET MVC with EF Core

Article • 04/10/2024

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:

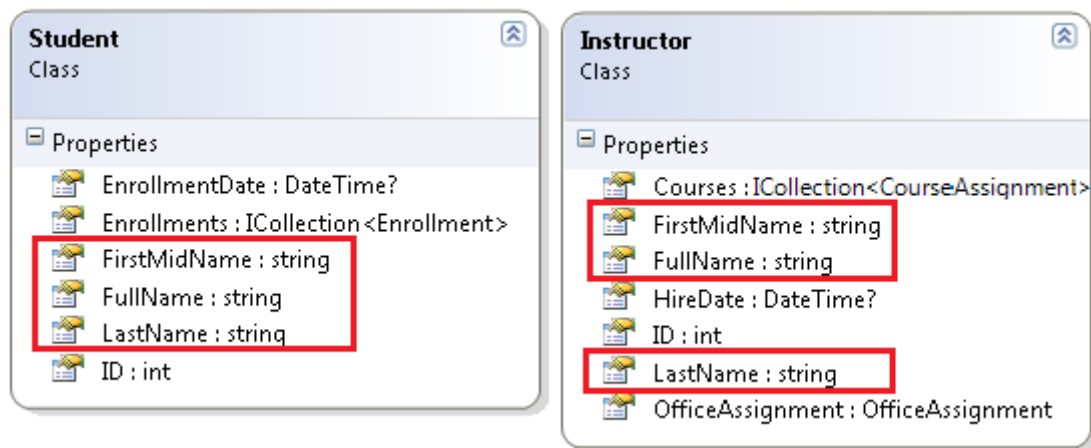
- ✓ Map inheritance to database
- ✓ Create the `Person` class
- ✓ Update `Instructor` and `Student`
- ✓ Add `Person` to the model
- ✓ Create and update migrations
- ✓ Test the implementation

## Prerequisites

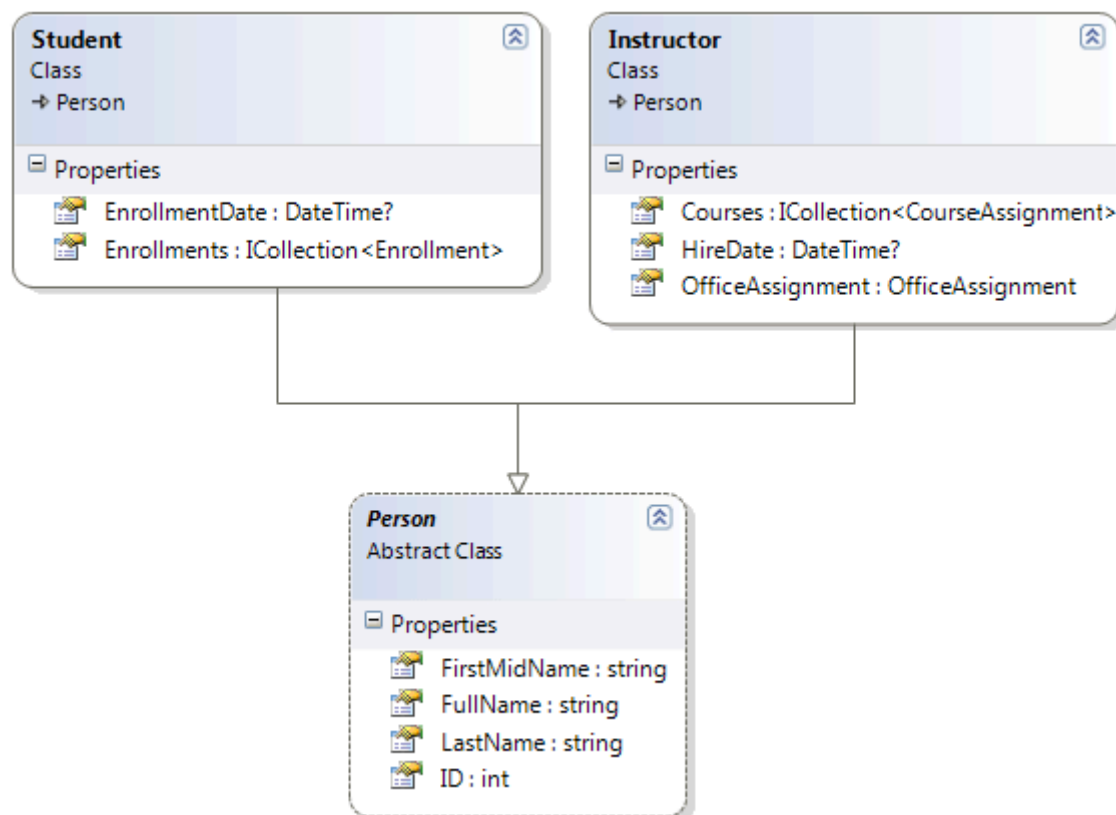
- [Handle Concurrency](#)

## Map inheritance to database

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

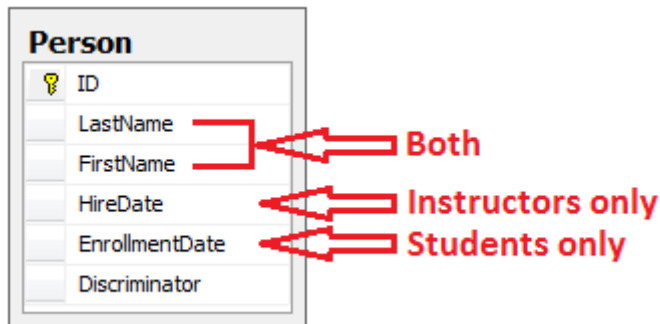


Suppose you want to eliminate the redundant code for the properties that are shared by the `Instructor` and `Student` entities. Or you want to write a service that can format names without caring whether the name came from an instructor or a student. You could create a `Person` base class that contains 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.

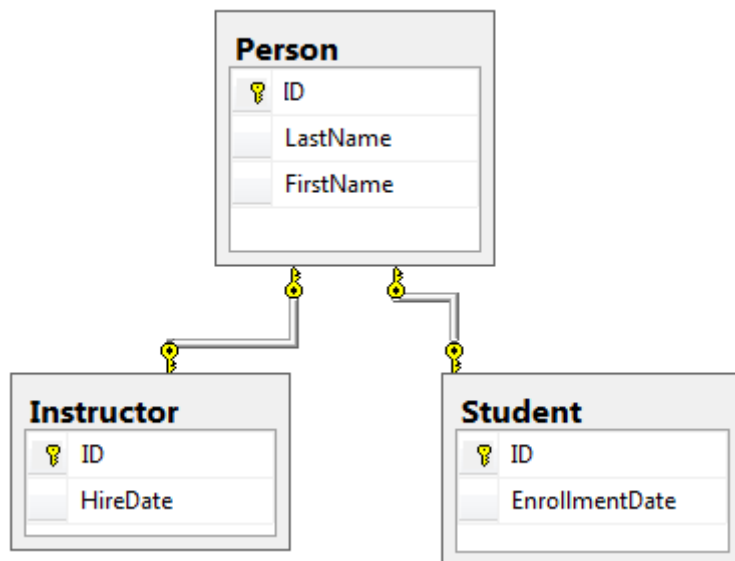


This pattern of generating an entity inheritance structure from a single database table is called *table-per-hierarchy (TPH)* inheritance.

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.

#### ⚠ Warning

Table-Per-Type (TPT) is not supported by EF Core 3.x, however it has been implemented in [EF Core 5.0](#).



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#

```
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#

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#

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#

```

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 CLI

```
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#

```

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 CLI
```

```
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.)

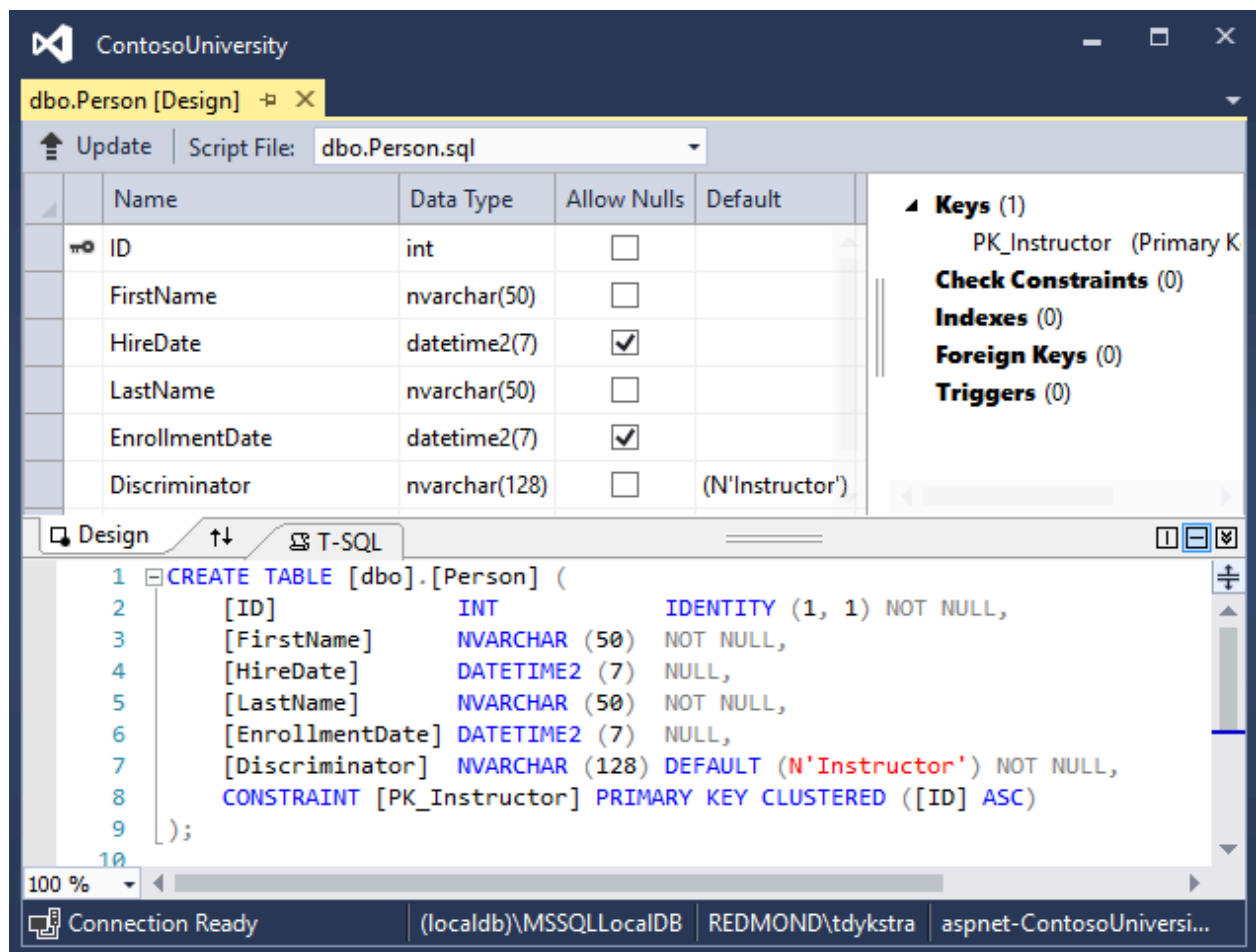
#### ⓘ Note

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 SSIX 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.

ContosoUniversity

dbo.Person [Data]

Max Rows: 1000

	ID	FirstName	HireDate	LastName	Enrollme...	Discriminator
1	1	Kim	3/11/1995...	Abercrombie	NULL	Instructor
2	2	Fadi	7/6/2002 ...	Fakhouri	NULL	Instructor
3	3	Roger	7/1/1998 ...	Harui	NULL	Instructor
4	4	Candace	1/15/2001...	Kapoor	NULL	Instructor
5	5	Roger	2/12/2004...	Zheng	NULL	Instructor
7	7	Nancy	8/17/2016...	Davolio	NULL	Instructor
8	8	Carson	NULL	Alexander	9/1/2010 ...	Student
9	9	Meredith	NULL	Alonso	9/1/2012 ...	Student
10	10	Arturo	NULL	Anand	9/1/2013 ...	Student
11	11	Gytis	NULL	Barzdukas	9/1/2012 ...	Student
12	12	Yan	NULL	Li	9/1/2012 ...	Student

## Get the code

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

# Additional resources

For more information about inheritance in Entity Framework Core, see [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.

**Next: Advanced topics**

# Tutorial: Learn about advanced scenarios - ASP.NET MVC with EF Core

Article • 04/10/2024

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](#)

## 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](#).
- Use the `Database.ExecuteNonQuery` 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#

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

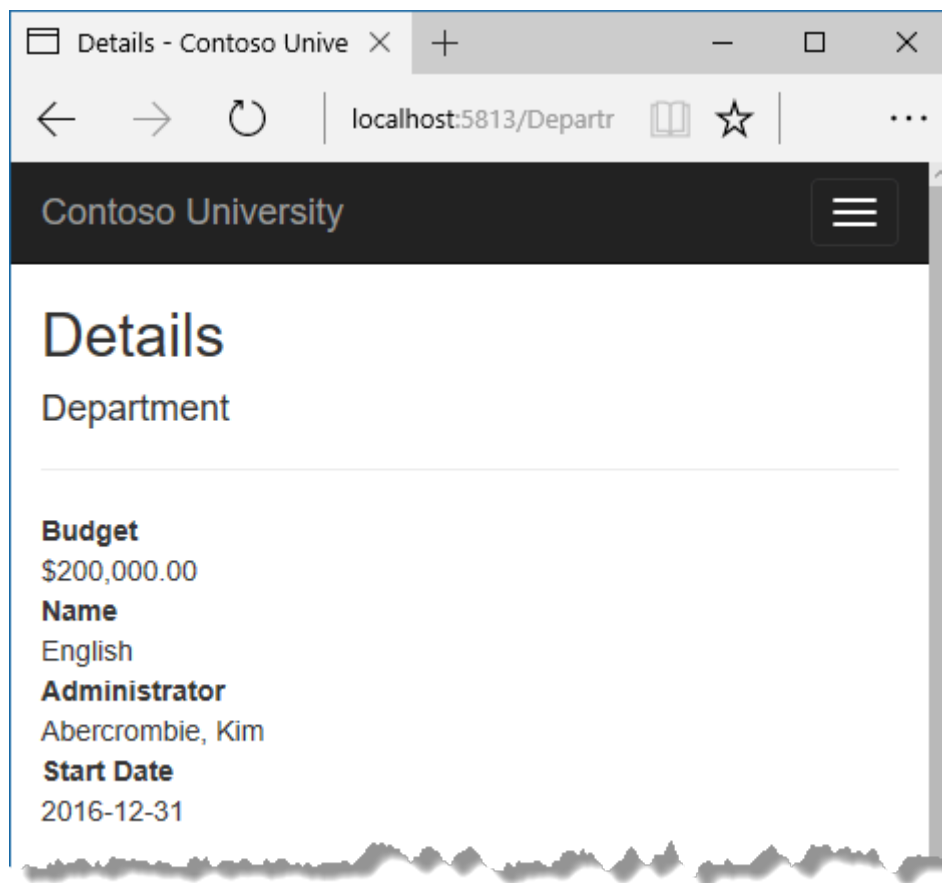
    string query = "SELECT * FROM Department WHERE DepartmentID = {0}";
    var department = await _context.Departments
        .FromSql(query, id)
        .Include(d => d.Administrator)
        .AsNoTracking()
        .FirstOrDefaultAsync();

    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 to write ADO.NET code and get the database connection from EF.

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

C#

```
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#

```
using System.Data.Common;
```

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

Enrollment Date	Students
9/1/2005	1
9/1/2010	1
9/1/2011	1
9/1/2012	3

## 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:

Update Course Credits

Enter a number to multiply every course's credits by:

[Back to list](#)

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

C#

```
public IActionResult UpdateCourseCredits()
{
```

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

C#

```
[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:

CSHTML

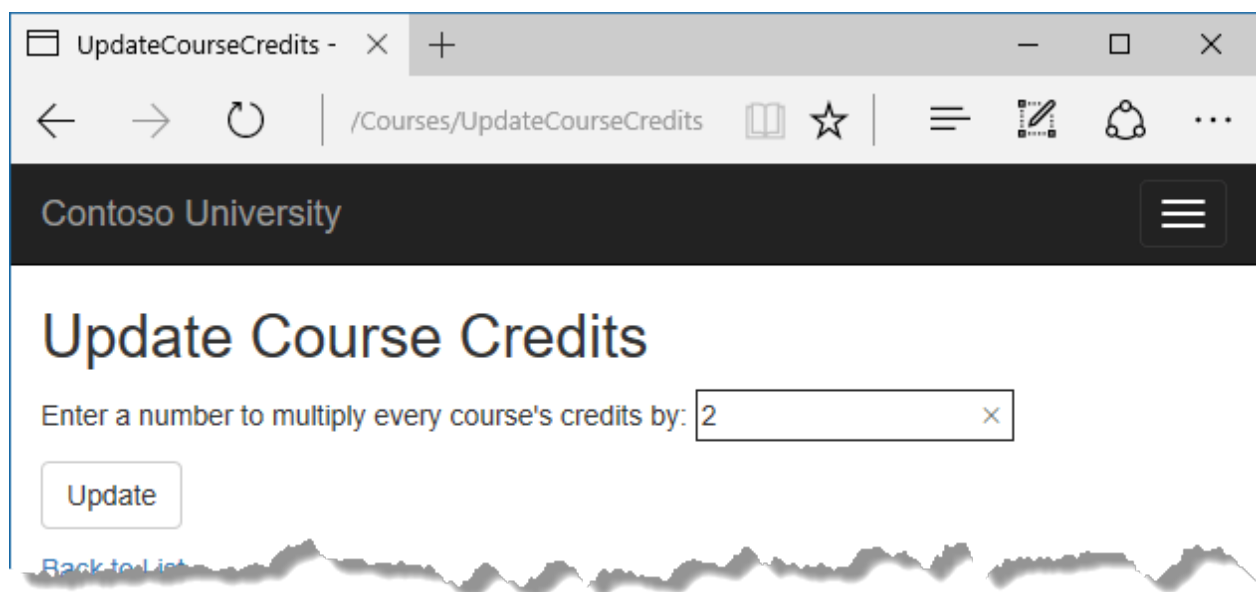
```
@{  
    ViewBag.Title = "UpdateCourseCredits";  
}  
  
<h2>Update Course Credits</h2>  
  
@if (ViewData["RowsAffected"] == null)  
{  
    <form asp-action="UpdateCourseCredits">  
        <div class="form-actions no-color">
```

```

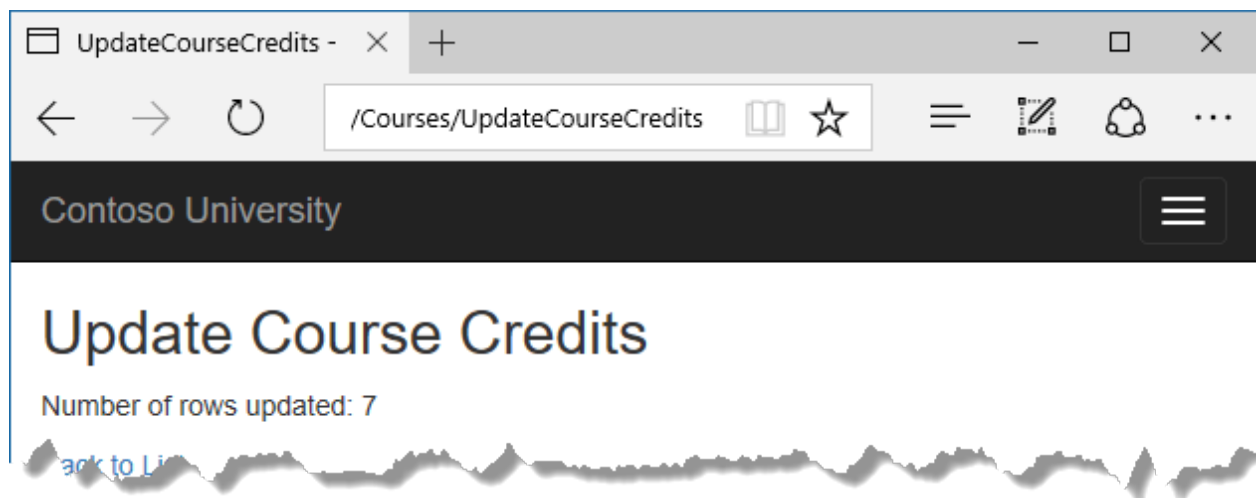
        <p>
            Enter a number to multiply every course's credits by:
            @Html.TextBox("multiplier")
        </p>
        <p>
            <input type="submit" value="Update" class="btn btn-default"
        />
        </p>
    </div>
</form>
}
@if (ViewData["RowsAffected"] != null)
{
    <p>
        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](#).

## 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:

```
Microsoft.EntityFrameworkCore.Database.Command: 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: 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](#).

Entity Framework Core implements an in-memory database provider that can be used for testing. For more information, see [Test with InMemory](#).

## Automatic change detection

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

- `DbContext.SaveChanges`
- `DbContext.Entry`
- `ChangeTracker.Entries`

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

`ChangeTracker.AutoDetectChangesEnabled` property. For example:

C#

```
_context.ChangeTracker.AutoDetectChangesEnabled = false;
```

## EF Core source code and development plans

The Entity Framework Core source is at <https://github.com/dotnet/efcore>. The EF Core repository contains nightly builds, issue tracking, feature specs, design meeting notes, and [the roadmap for future development](#). 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](#) command. See the [getting-started tutorial](#).

## Use dynamic LINQ to simplify code

The [third tutorial in this series](#) shows how to write LINQ code by hard-coding column names in a `switch` statement. With two columns to choose from, this works fine, but if you have many columns the code could get verbose. To solve that problem, you can use the `EF.Property` method to specify the name of the property as a string. To try out this approach, replace the `Index` method in the `StudentsController` with the following code.

C#

```
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 CLI
```

```
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.

## Get the code

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

## Additional resources

For more information about EF Core, see the [Entity Framework Core documentation](#). A book is also available: [Entity Framework Core in Action](#) .

For information on how to deploy a web app, see [Host and deploy ASP.NET Core](#).

For information about other topics related to ASP.NET Core MVC, such as authentication and authorization, see [Overview of ASP.NET Core](#).

## 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](#).



# ASP.NET Core fundamentals overview

Article • 11/14/2024

## Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article provides an overview of the fundamentals for building ASP.NET Core apps, including dependency injection (DI), configuration, middleware, and more.

For Blazor fundamentals guidance, which adds to or supersedes the guidance in this node, see [ASP.NET Core Blazor fundamentals](#).

## Program.cs

ASP.NET Core apps created with the web templates contain the application startup code in the `Program.cs` file. The `Program.cs` file is where:

- Services required by the app are configured.
- The app's request handling pipeline is defined as a series of [middleware components](#).

The following app startup code supports:

- [Razor Pages](#)
- [MVC controllers with views](#)
- [Web API with controllers](#)
- [Minimal web APIs](#)

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();
```

```
// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

app.UseAuthorization();

app.MapGet("/hi", () => "Hello!");

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

## Dependency injection (services)

ASP.NET Core includes [dependency injection \(DI\)](#) that makes configured services available throughout an app. Services are added to the DI container with [WebApplicationBuilder.Services](#), `builder.Services` in the preceding code. When the [WebApplicationBuilder](#) is instantiated, many [framework-provided services](#) are added. `builder` is a `WebApplicationBuilder` in the following code:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();
```

In the preceding highlighted code, `builder` has configuration, logging, and [many other services](#) added to the DI container.

The following code adds Razor Pages, MVC controllers with views, and a custom [DbContext](#) to the DI container:

C#

```
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Data;
```

```

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RPMovieConte
xt"))));

var app = builder.Build();

```

Services are typically resolved from DI using constructor injection. The DI framework provides an instance of this service at runtime.

The following code uses constructor injection to resolve the database context and logger from DI:

```

C#

public class IndexModel : PageModel
{
    private readonly RazorPagesMovieContext _context;
    private readonly ILogger<IndexModel> _logger;

    public IndexModel(RazorPagesMovieContext context, ILogger<IndexModel>
logger)
    {
        _context = context;
        _logger = logger;
    }

    public IList<Movie> Movie { get;set; }

    public async Task OnGetAsync()
    {
        _logger.LogInformation("IndexModel OnGetAsync.");
        Movie = await _context.Movie.ToListAsync();
    }
}

```

## Middleware

The request handling pipeline is composed as a series of middleware components. Each component performs operations on an [HttpContext](#) and either invokes the next middleware in the pipeline or terminates the request.



By convention, a middleware component is added to the pipeline by invoking a `Use{Feature}` extension method. Middleware added to the app is highlighted in the following code:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

app.UseAuthorization();

app.MapGet("/hi", () => "Hello!");

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

For more information, see [ASP.NET Core Middleware](#).

## Host

On startup, an ASP.NET Core app builds a *host*. The host encapsulates all of the app's resources, such as:

- An HTTP server implementation
- Middleware components
- Logging
- Dependency injection (DI) services
- Configuration

There are three different hosts capable of running an ASP.NET Core app:

- [ASP.NET Core WebApplication](#), also known as the [Minimal Host](#)
- [.NET Generic Host](#) combined with ASP.NET Core's [ConfigureWebHostDefaults](#)
- [ASP.NET Core WebHost](#)

The ASP.NET Core [WebApplication](#) and [WebApplicationBuilder](#) types are recommended and used in all the ASP.NET Core templates. `WebApplication` behaves similarly to the .NET Generic Host and exposes many of the same interfaces but requires less callbacks to configure. The ASP.NET Core [WebHost](#) is available only for backward compatibility.

The following example instantiates a `WebApplication`:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();
```

The [WebApplicationBuilder.Build](#) method configures a host with a set of default options, such as:

- Use [Kestrel](#) as the web server and enable IIS integration.
- Load [configuration](#) from `appsettings.json`, environment variables, command line arguments, and other configuration sources.
- Send logging output to the console and debug providers.

## Non-web scenarios

The Generic Host allows other types of apps to use cross-cutting framework extensions, such as logging, dependency injection (DI), configuration, and app lifetime management. For more information, see [.NET Generic Host in ASP.NET Core](#) and [Background tasks with hosted services in ASP.NET Core](#).

## Servers

An ASP.NET Core app uses an HTTP server implementation to listen for HTTP requests. The server surfaces requests to the app as a set of [request features](#) composed into an `HttpContext`.

ASP.NET Core provides the following server implementations:

- *Kestrel* is a cross-platform web server. Kestrel is often run in a reverse proxy configuration using [IIS](#). In ASP.NET Core 2.0 or later, Kestrel can be run as a public-facing edge server exposed directly to the Internet.
- *IIS HTTP Server* is a server for Windows that uses IIS. With this server, the ASP.NET Core app and IIS run in the same process.
- *HTTP.sys* is a server for Windows that isn't used with IIS.

For more information, see [Web server implementations in ASP.NET Core](#).

## Configuration

ASP.NET Core provides a [configuration](#) framework that gets settings as name-value pairs from an ordered set of configuration providers. Built-in configuration providers are available for a variety of sources, such as `.json` files, `.xml` files, environment variables, and command-line arguments. Write custom configuration providers to support other sources.

By [default](#), ASP.NET Core apps are configured to read from `appsettings.json`, environment variables, the command line, and more. When the app's configuration is loaded, values from environment variables override values from `appsettings.json`.

For managing confidential configuration data such as passwords, .NET Core provides the [Secret Manager](#). For production secrets, we recommend [Azure Key Vault](#).

For more information, see [Configuration in ASP.NET Core](#).

## Environments

Execution environments, such as `Development`, `Staging`, and `Production`, are available in ASP.NET Core. Specify the environment an app is running in by setting the `ASPNETCORE_ENVIRONMENT` environment variable. ASP.NET Core reads that environment variable at app startup and stores the value in an `IWebHostEnvironment` implementation. This implementation is available anywhere in an app via dependency injection (DI).

The following example configures the exception handler and [HTTP Strict Transport Security Protocol \(HSTS\)](#) middleware when *not* running in the `Development` environment:

```
C#
```

```

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

app.UseAuthorization();

app.MapGet("/hi", () => "Hello!");

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();

```

For more information, see [Use multiple environments in ASP.NET Core](#).

## Logging

ASP.NET Core supports a logging API that works with a variety of built-in and third-party logging providers. Available providers include:

- Console
- Debug
- Event Tracing on Windows
- Windows Event Log
- TraceSource
- Azure App Service
- Azure Application Insights

To create logs, resolve an `ILogger<TCategoryName>` service from dependency injection (DI) and call logging methods such as [LogInformation](#). For example:

C#

```

public class IndexModel : PageModel
{
    private readonly RazorPagesMovieContext _context;
    private readonly ILogger<IndexModel> _logger;

    public IndexModel(RazorPagesMovieContext context, ILogger<IndexModel>
logger)
    {
        _context = context;
        _logger = logger;
    }

    public IList<Movie> Movie { get;set; }

    public async Task OnGetAsync()
    {
        _logger.LogInformation("IndexModel OnGetAsync.");
        Movie = await _context.Movie.ToListAsync();
    }
}

```

For more information, see [Logging in .NET Core and ASP.NET Core](#).

## Routing

A *route* is a URL pattern that is mapped to a handler. The handler is typically a Razor page, an action method in an MVC controller, or a middleware. ASP.NET Core routing gives you control over the URLs used by your app.

The following code, generated by the ASP.NET Core web application template, calls [UseRouting](#):

```

C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

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

```

```
app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

For more information, see [Routing in ASP.NET Core](#).

## Error handling

ASP.NET Core has built-in features for handling errors, such as:

- A developer exception page
- Custom error pages
- Static status code pages
- Startup exception handling

For more information, see [Handle errors in ASP.NET Core](#).

## Make HTTP requests

An implementation of `IHttpClientFactory` is available for creating `HttpClient` instances. The factory:

- Provides a central location for naming and configuring logical `HttpClient` instances. For example, register and configure a *github* client for accessing GitHub. Register and configure a default client for other purposes.
- Supports registration and chaining of multiple delegating handlers to build an outgoing request middleware pipeline. This pattern is similar to ASP.NET Core's inbound middleware pipeline. The pattern provides a mechanism to manage cross-cutting concerns for HTTP requests, including caching, error handling, serialization, and logging.
- Integrates with *Polly*, a popular third-party library for transient fault handling.
- Manages the pooling and lifetime of underlying `HttpClientHandler` instances to avoid common DNS problems that occur when managing `HttpClient` lifetimes manually.
- Adds a configurable logging experience via [ILogger](#) for all requests sent through clients created by the factory.

For more information, see [Make HTTP requests using IHttpClientFactory in ASP.NET Core](#).

## Content root

The content root is the base path for:

- The executable hosting the app (.exe).
- Compiled assemblies that make up the app (.dll).
- Content files used by the app, such as:
  - Razor files (.cshtml, .razor)
  - Configuration files (.json, .xml)
  - Data files (.db)
- The [Web root](#), typically the *wwwroot* folder.

During development, the content root defaults to the project's root directory. This directory is also the base path for both the app's content files and the [Web root](#). Specify a different content root by setting its path when [building the host](#). For more information, see [Content root](#).

## Web root

The web root is the base path for public, static resource files, such as:

- Stylesheets (.css)
- JavaScript (.js)
- Images (.png, .jpg)

By default, static files are served only from the web root directory and its sub-directories. The web root path defaults to *{content root}/wwwroot*. Specify a different web root by setting its path when [building the host](#). For more information, see [Web root](#).

Prevent publishing files in *wwwroot* with the [<Content> project item](#) in the project file. The following example prevents publishing content in *wwwroot/local* and its sub-directories:

XML

```
<ItemGroup>
  <Content Update="wwwroot\local\**\*.*" CopyToPublishDirectory="Never" />
</ItemGroup>
```

In Razor `.cshtml` files, `~/` points to the web root. A path beginning with `~/` is referred to as a *virtual path*.

For more information, see [Static files in ASP.NET Core](#).

## Additional resources

- [WebApplicationBuilder source code](#) 



# App startup in ASP.NET Core

Article • 12/12/2024

## Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

By [Rick Anderson](#) 

ASP.NET Core apps created with the web templates contain the application startup code in the `Program.cs` file.

For Blazor startup guidance, which adds to or supersedes the guidance in this article, see [ASP.NET Core Blazor startup](#).

The following app startup code supports several app types:

- [Blazor Web Apps](#)
- [Razor Pages](#)
- [MVC controllers with views](#)
- [Web API with controllers](#)
- [Minimal APIs](#)

C#

```
using WebAll.Components;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
```

```

}

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

app.UseAuthorization();

app.MapGet("/hi", () => "Hello!");

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();

app.UseAntiforgery();

app.Run();

```

Apps that use [EventSource](#) can measure the startup time to understand and optimize startup performance. The [ServerReady](#) [event](#) in [Microsoft.AspNetCore.Hosting](#) represents the point where the server is ready to respond to requests.

## Extend Startup with startup filters

Use [IStartupFilter](#):

- To configure middleware at the beginning or end of an app's middleware pipeline without an explicit call to `Use{Middleware}`. Use `IStartupFilter` to add defaults to the beginning of the pipeline without explicitly registering the default middleware. `IStartupFilter` allows a different component to call `Use{Middleware}` on behalf of the app author.
- To create a pipeline of `Configure` methods. [IStartupFilter.Configure](#) can set a middleware to run before or after middleware added by libraries.

An `IStartupFilter` implementation implements [Configure](#), which receives and returns an `Action<IApplicationBuilder>`. An [IApplicationBuilder](#) defines a class to configure an app's request pipeline. For more information, see [Create a middleware pipeline with IApplicationBuilder](#).

Each `IStartupFilter` implementation can add one or more middlewares in the request pipeline. The filters are invoked in the order they were added to the service container. Filters can add middleware before or after passing control to the next filter, thus they append to the beginning or end of the app pipeline.

The following example demonstrates how to register a middleware with `IStartupFilter`. The `RequestSetOptionsMiddleware` middleware sets an options value from a query string parameter:

C#

```
public class RequestSetOptionsMiddleware
{
    private readonly RequestDelegate _next;

    public RequestSetOptionsMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    // Test with https://localhost:5001/Privacy/?option=Hello
    public async Task Invoke(HttpContext httpContext)
    {
        var option = httpContext.Request.Query["option"];

        if (!string.IsNullOrEmpty(option))
        {
            httpContext.Items["option"] = WebUtility.HtmlEncode(option);
        }

        await _next(httpContext);
    }
}
```

The `RequestSetOptionsMiddleware` is configured in the `RequestSetOptionsStartupFilter` class:

C#

```
public class RequestSetOptionsStartupFilter : IStartupFilter
{
    public Action<IApplicationBuilder> Configure(Action<IApplicationBuilder> next)
    {
        return builder =>
        {
            builder.UseMiddleware<RequestSetOptionsMiddleware>();
            next(builder);
        };
    }
}
```

The `IStartupFilter` implementation is registered in `Program.cs`:

C#

```

using WebStartup.Middleware;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddTransient<IStartupFilter,
    RequestSetOptionsStartupFilter>();

var app = builder.Build();

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

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

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();

```

When a query string parameter for `option` is provided, the middleware processes the value assignment before the ASP.NET Core middleware renders the response:

CSHTML

```

@page
@model PrivacyModel
@{
    ViewData["Title"] = "Privacy Policy";
}
<h1>@ViewData["Title"]</h1>

<p> Append query string ?option=hello</p>
Option String: @HttpContext.Items["option"];

```

Middleware execution order is set by the order of `IStartupFilter` registrations:

- Multiple `IStartupFilter` implementations might interact with the same objects. If ordering is important, order their `IStartupFilter` service registrations to match the order that their middlewares should run.

- Libraries can add middleware with one or more `IStartupFilter` implementations that run before or after other app middleware registered with `IStartupFilter`. To invoke an `IStartupFilter` middleware before a middleware added by a library's `IStartupFilter`:
  - Position the service registration before the library is added to the service container.
  - To invoke afterward, position the service registration after the library is added.

You can't extend the ASP.NET Core app when you override `Configure`. For more information, see [this GitHub issue](#).

## Add configuration at startup from an external assembly

An `IHostingStartup` implementation allows adding enhancements to an app at startup from an external assembly outside of the app's `Program.cs` file. For more information, see [Use hosting startup assemblies in ASP.NET Core](#).

## Startup, ConfigureServices, and Configure

For information on using the `ConfigureServices` and `Configure` methods with the minimal hosting model, see:

- [Use Startup with the minimal hosting model](#)
- The ASP.NET Core 5.0 version of this article:
  - [The ConfigureServices method](#)
  - [The Configure method](#)

# Dependency injection in ASP.NET Core

Article • 09/18/2024

## Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

By [Kirk Larkin](#), [Steve Smith](#), and [Brandon Dahler](#)

ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving [Inversion of Control \(IoC\)](#) between classes and their dependencies.

For more information specific to dependency injection within MVC controllers, see [Dependency injection into controllers in ASP.NET Core](#).

For information on using dependency injection in applications other than web apps, see [Dependency injection in .NET](#).

For more information on dependency injection of options, see [Options pattern in ASP.NET Core](#).

This topic provides information on dependency injection in ASP.NET Core. The primary documentation on using dependency injection is contained in [Dependency injection in .NET](#).

For Blazor DI guidance, which adds to or supersedes the guidance in this article, see [ASP.NET Core Blazor dependency injection](#).

[View or download sample code](#) [\(how to download\)](#)

## Overview of dependency injection

A *dependency* is an object that another object depends on. Examine the following `MyDependency` class with a `WriteMessage` method that other classes depend on:

```
C#
```

```

public class MyDependency
{
    public void WriteMessage(string message)
    {
        Console.WriteLine($"MyDependency.WriteMessage called. Message:
{message}");
    }
}

```

A class can create an instance of the `MyDependency` class to make use of its `WriteMessage` method. In the following example, the `MyDependency` class is a dependency of the `IndexModel` class:

C#

```

public class IndexModel : PageModel
{
    private readonly MyDependency _dependency = new MyDependency();

    public void OnGet()
    {
        _dependency.WriteMessage("IndexModel.OnGet");
    }
}

```

The class creates and directly depends on the `MyDependency` class. Code dependencies, such as in the previous example, are problematic and should be avoided for the following reasons:

- To replace `MyDependency` with a different implementation, the `IndexModel` class must be modified.
- If `MyDependency` has dependencies, they must also be configured by the `IndexModel` class. In a large project with multiple classes depending on `MyDependency`, the configuration code becomes scattered across the app.
- This implementation is difficult to unit test.

Dependency injection addresses these problems through:

- The use of an interface or base class to abstract the dependency implementation.
- Registration of the dependency in a service container. ASP.NET Core provides a built-in service container, `IServiceProvider`. Services are typically registered in the app's `Program.cs` file.
- *Injection* of the service into the constructor of the class where it's used. The framework takes on the responsibility of creating an instance of the dependency

and disposing of it when it's no longer needed.

In the [sample app](#), the `IMyDependency` interface defines the `WriteMessage` method:

```
C#  
  
public interface IMyDependency  
{  
    void WriteMessage(string message);  
}
```

This interface is implemented by a concrete type, `MyDependency`:

```
C#  
  
public class MyDependency : IMyDependency  
{  
    public void WriteMessage(string message)  
    {  
        Console.WriteLine($"MyDependency.WriteMessage Message: {message}");  
    }  
}
```

The sample app registers the `IMyDependency` service with the concrete type `MyDependency`. The `AddScoped` method registers the service with a scoped lifetime, the lifetime of a single request. [Service lifetimes](#) are described later in this topic.

```
C#  
  
using DependencyInjectionSample.Interfaces;  
using DependencyInjectionSample.Services;  
  
var builder = WebApplication.CreateBuilder(args);  
  
builder.Services.AddRazorPages();  
  
builder.Services.AddScoped<IMyDependency, MyDependency>();  
  
var app = builder.Build();
```

In the sample app, the `IMyDependency` service is requested and used to call the `WriteMessage` method:

```
C#  
  
public class Index2Model : PageModel  
{
```



```

private readonly IMyDependency _myDependency;

public Index2Model(IMyDependency myDependency)
{
    _myDependency = myDependency;
}

public void OnGet()
{
    _myDependency.WriteMessage("Index2Model.OnGet");
}
}

```

By using the DI pattern, the controller or Razor Page:

- Doesn't use the concrete type `MyDependency`, only the `IMyDependency` interface it implements. That makes it easy to change the implementation without modifying the controller or Razor Page.
- Doesn't create an instance of `MyDependency`, it's created by the DI container.

The implementation of the `IMyDependency` interface can be improved by using the built-in logging API:

C#

```

public class MyDependency2 : IMyDependency
{
    private readonly ILogger<MyDependency2> _logger;

    public MyDependency2(ILogger<MyDependency2> logger)
    {
        _logger = logger;
    }

    public void WriteMessage(string message)
    {
        _logger.LogInformation($"MyDependency2.WriteMessage Message: {message}");
    }
}

```

The updated `Program.cs` registers the new `IMyDependency` implementation:

C#

```

using DependencyInjectionSample.Interfaces;
using DependencyInjectionSample.Services;

var builder = WebApplication.CreateBuilder(args);

```

```
builder.Services.AddRazorPages();

builder.Services.AddScoped<IMyDependency, MyDependency2>();

var app = builder.Build();
```

`MyDependency2` depends on `ILogger<TCategoryName>`, which it requests in the constructor. `ILogger<TCategoryName>` is a [framework-provided service](#).

It's not unusual to use dependency injection in a chained fashion. Each requested dependency in turn requests its own dependencies. The container resolves the dependencies in the graph and returns the fully resolved service. The collective set of dependencies that must be resolved is typically referred to as a *dependency tree*, *dependency graph*, or *object graph*.

The container resolves `ILogger<TCategoryName>` by taking advantage of [\(generic\) open types](#), eliminating the need to register every [\(generic\) constructed type](#).

In dependency injection terminology, a service:

- Is typically an object that provides a service to other objects, such as the `IMyDependency` service.
- Is not related to a web service, although the service may use a web service.

The framework provides a robust [logging](#) system. The `IMyDependency` implementations shown in the preceding examples were written to demonstrate basic DI, not to implement logging. Most apps shouldn't need to write loggers. The following code demonstrates using the default logging, which doesn't require any services to be registered:

```
C#

public class AboutModel : PageModel
{
    private readonly ILogger _logger;

    public AboutModel(ILogger<AboutModel> logger)
    {
        _logger = logger;
    }

    public string Message { get; set; } = string.Empty;

    public void OnGet()
    {
        Message = $"About page visited at {DateTime.UtcNow.ToLongTimeString()}";
    }
}
```

```
        _logger.LogInformation(Message);  
    }  
}
```

Using the preceding code, there is no need to update `Program.cs`, because [logging](#) is provided by the framework.

## Register groups of services with extension methods

The ASP.NET Core framework uses a convention for registering a group of related services. The convention is to use a single `Add{GROUP_NAME}` extension method to register all of the services required by a framework feature. For example, the [AddControllers](#) extension method registers the services required for MVC controllers.

The following code is generated by the Razor Pages template using individual user accounts and shows how to add additional services to the container using the extension methods [AddDbContext](#) and [AddDefaultIdentity](#):

C#

```
using DependencyInjectionSample.Data;  
using Microsoft.AspNetCore.Identity;  
using Microsoft.EntityFrameworkCore;  
  
var builder = WebApplication.CreateBuilder(args);  
  
var connectionString =  
builder.Configuration.GetConnectionString("DefaultConnection");  
builder.Services.AddDbContext<ApplicationDbContext>(options =>  
    options.UseSqlServer(connectionString));  
builder.Services.AddDatabaseDeveloperPageExceptionFilter();  
  
builder.Services.AddDefaultIdentity<IdentityUser>(options =>  
    options.SignIn.RequireConfirmedAccount = true)  
    .AddEntityFrameworkStores<ApplicationDbContext>();  
builder.Services.AddRazorPages();  
  
var app = builder.Build();
```

Consider the following which registers services and configures options:

C#

```
using ConfigSample.Options;  
using Microsoft.Extensions.DependencyInjection.ConfigSample.Options;
```

```

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<PositionOptions>(
    builder.Configuration.GetSection(PositionOptions.Position));
builder.Services.Configure<ColorOptions>(
    builder.Configuration.GetSection(ColorOptions.Color));

builder.Services.AddScoped<IMyDependency, MyDependency>();
builder.Services.AddScoped<IMyDependency2, MyDependency2>();

var app = builder.Build();

```

Related groups of registrations can be moved to an extension method to register services. For example, the configuration services are added to the following class:

C#

```

using ConfigSample.Options;
using Microsoft.Extensions.Configuration;

namespace Microsoft.Extensions.DependencyInjection
{
    public static class MyConfigServiceCollectionExtensions
    {
        public static IServiceCollection AddConfig(
            this IServiceCollection services, IConfiguration config)
        {
            services.Configure<PositionOptions>(
                config.GetSection(PositionOptions.Position));
            services.Configure<ColorOptions>(
                config.GetSection(ColorOptions.Color));

            return services;
        }

        public static IServiceCollection AddMyDependencyGroup(
            this IServiceCollection services)
        {
            services.AddScoped<IMyDependency, MyDependency>();
            services.AddScoped<IMyDependency2, MyDependency2>();

            return services;
        }
    }
}

```

The remaining services are registered in a similar class. The following code uses the new extension methods to register the services:

C#

```
using Microsoft.Extensions.DependencyInjection.ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddConfig(builder.Configuration)
    .AddMyDependencyGroup();

builder.Services.AddRazorPages();

var app = builder.Build();
```

**Note:** Each `services.Add{GROUP_NAME}` extension method adds and potentially configures services. For example, [AddControllersWithViews](#) adds the services MVC controllers with views require, and [AddRazorPages](#) adds the services Razor Pages requires.

## Service lifetimes

See [Service lifetimes](#) in [Dependency injection in .NET](#)

To use scoped services in middleware, use one of the following approaches:

- Inject the service into the middleware's `Invoke` or `InvokeAsync` method. Using [constructor injection](#) throws a runtime exception because it forces the scoped service to behave like a singleton. The sample in the [Lifetime and registration options](#) section demonstrates the `InvokeAsync` approach.
- Use [Factory-based middleware](#). Middleware registered using this approach is activated per client request (connection), which allows scoped services to be injected into the middleware's constructor.

For more information, see [Write custom ASP.NET Core middleware](#).

## Service registration methods

See [Service registration methods](#) in [Dependency injection in .NET](#)

It's common to use multiple implementations when [mocking types for testing](#).

Registering a service with only an implementation type is equivalent to registering that service with the same implementation and service type. This is why multiple implementations of a service cannot be registered using the methods that don't take an explicit service type. These methods can register multiple *instances* of a service, but they will all have the same *implementation* type.

Any of the above service registration methods can be used to register multiple service instances of the same service type. In the following example, `AddSingleton` is called twice with `IMyDependency` as the service type. The second call to `AddSingleton` overrides the previous one when resolved as `IMyDependency` and adds to the previous one when multiple services are resolved via `IEnumerable<IMyDependency>`. Services appear in the order they were registered when resolved via `IEnumerable<{SERVICE}>`.

C#

```
services.AddSingleton<IMyDependency, MyDependency>();
services.AddSingleton<IMyDependency, DifferentDependency>();

public class MyService
{
    public MyService(IMyDependency myDependency,
        IEnumerable<IMyDependency> myDependencies)
    {
        Trace.Assert(myDependency is DifferentDependency);

        var dependencyArray = myDependencies.ToArray();
        Trace.Assert(dependencyArray[0] is MyDependency);
        Trace.Assert(dependencyArray[1] is DifferentDependency);
    }
}
```

## Keyed services

*Keyed services* refers to a mechanism for registering and retrieving Dependency Injection (DI) services using keys. A service is associated with a key by calling `AddKeyedSingleton` (or `AddKeyedScoped` or `AddKeyedTransient`) to register it. Access a registered service by specifying the key with the `[FromKeyedServices]` attribute. The following code shows how to use keyed services:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.SignalR;

var builder = WebApplication.CreateBuilder(args);
```

```

builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");
builder.Services.AddControllers();

var app = builder.Build();

app.MapGet("/big", ([FromKeyedServices("big")] ICache bigCache) =>
bigCache.Get("date"));
app.MapGet("/small", ([FromKeyedServices("small")] ICache smallCache) =>
smallCache.Get("date"));

app.MapControllers();

app.Run();

public interface ICache
{
    object Get(string key);
}
public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}

[ApiController]
[Route("/cache")]
public class CustomServicesApiController : Controller
{
    [HttpGet("big-cache")]
    public ActionResult<object> GetOk([FromKeyedServices("big")] ICache
cache)
    {
        return cache.Get("data-mvc");
    }
}

public class MyHub : Hub
{
    public void Method([FromKeyedServices("small")] ICache cache)
    {
        Console.WriteLine(cache.Get("signalr"));
    }
}

```

## Keyed services in Middleware

Middleware supports Keyed services in both the constructor and the

`Invoke` / `InvokeAsync` method:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddKeyedSingleton<MySingletonClass>("test");
builder.Services.AddKeyedScoped<MyScopedClass>("test2");

var app = builder.Build();
app.UseMiddleware<MyMiddleware>();
app.Run();

internal class MyMiddleware
{
    private readonly RequestDelegate _next;

    public MyMiddleware(RequestDelegate next,
        [FromKeyedServices("test")] MySingletonClass service)
    {
        _next = next;
    }

    public Task Invoke(HttpContext context,
        [FromKeyedServices("test2")]
        MyScopedClass scopedService) => _next(context);
}
```

For more information on creating Middleware, see [Write custom ASP.NET Core middleware](#)

## Constructor injection behavior

See [Constructor injection behavior](#) in [Dependency injection in .NET](#)

## Entity Framework contexts

By default, Entity Framework contexts are added to the service container using the [scoped lifetime](#) because web app database operations are normally scoped to the client request. To use a different lifetime, specify the lifetime by using an [AddDbContext](#) overload. Services of a given lifetime shouldn't use a database context with a lifetime that's shorter than the service's lifetime.

## Lifetime and registration options



To demonstrate the difference between service lifetimes and their registration options, consider the following interfaces that represent a task as an operation with an identifier, `OperationId`. Depending on how the lifetime of an operation's service is configured for the following interfaces, the container provides either the same or different instances of the service when requested by a class:

C#

```
public interface IOperation
{
    string OperationId { get; }
}

public interface IOperationTransient : IOperation { }
public interface IOperationScoped : IOperation { }
public interface IOperationSingleton : IOperation { }
```

The following `Operation` class implements all of the preceding interfaces. The `Operation` constructor generates a GUID and stores the last 4 characters in the `OperationId` property:

C#

```
public class Operation : IOperationTransient, IOperationScoped,
    IOperationSingleton
{
    public Operation()
    {
        OperationId = Guid.NewGuid().ToString()[^4..];
    }

    public string OperationId { get; }
}
```

The following code creates multiple registrations of the `Operation` class according to the named lifetimes:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddTransient<IOperationTransient, Operation>();
builder.Services.AddScoped<IOperationScoped, Operation>();
builder.Services.AddSingleton<IOperationSingleton, Operation>();

var app = builder.Build();
```

```

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

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

app.UseMyMiddleware();
app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();

```

The sample app demonstrates object lifetimes both within and between requests. The `IndexModel` and the middleware request each kind of `IOperation` type and log the `OperationId` for each:

C#

```

public class IndexModel : PageModel
{
    private readonly ILogger _logger;
    private readonly IOperationTransient _transientOperation;
    private readonly IOperationSingleton _singletonOperation;
    private readonly IOperationScoped _scopedOperation;

    public IndexModel(ILogger<IndexModel> logger,
                     IOperationTransient transientOperation,
                     IOperationScoped scopedOperation,
                     IOperationSingleton singletonOperation)
    {
        _logger = logger;
        _transientOperation = transientOperation;
        _scopedOperation = scopedOperation;
        _singletonOperation = singletonOperation;
    }

    public void OnGet()
    {
        _logger.LogInformation("Transient: " +
                               _transientOperation.OperationId);
        _logger.LogInformation("Scoped: " +
                               _scopedOperation.OperationId);
        _logger.LogInformation("Singleton: " +
                               _singletonOperation.OperationId);
    }
}

```

```
}  
}
```

Similar to the `IndexModel`, the middleware resolves the same services:

C#

```
public class MyMiddleware  
{  
    private readonly RequestDelegate _next;  
    private readonly ILogger _logger;  
  
    private readonly IOperationSingleton _singletonOperation;  
  
    public MyMiddleware(RequestDelegate next, ILogger<MyMiddleware> logger,  
        IOperationSingleton singletonOperation)  
    {  
        _logger = logger;  
        _singletonOperation = singletonOperation;  
        _next = next;  
    }  
  
    public async Task InvokeAsync(HttpContext context,  
        IOperationTransient transientOperation, IOperationScoped  
scopedOperation)  
    {  
        _logger.LogInformation("Transient: " +  
transientOperation.OperationId);  
        _logger.LogInformation("Scoped: " + scopedOperation.OperationId);  
        _logger.LogInformation("Singleton: " +  
_singletonOperation.OperationId);  
  
        await _next(context);  
    }  
}  
  
public static class MyMiddlewareExtensions  
{  
    public static IApplicationBuilder UseMyMiddleware(this  
IApplicationBuilder builder)  
    {  
        return builder.UseMiddleware<MyMiddleware>();  
    }  
}
```

Scoped and transient services must be resolved in the `InvokeAsync` method:

C#

```
public async Task InvokeAsync(HttpContext context,
    IOperationTransient transientOperation, IOperationScoped
scopedOperation)
{
    _logger.LogInformation("Transient: " + transientOperation.OperationId);
    _logger.LogInformation("Scoped: " + scopedOperation.OperationId);
    _logger.LogInformation("Singleton: " + _singletonOperation.OperationId);

    await _next(context);
}
```

The logger output shows:

- *Transient* objects are always different. The transient `OperationId` value is different in the `IndexModel` and in the middleware.
- *Scoped* objects are the same for a given request but differ across each new request.
- *Singleton* objects are the same for every request.

To reduce the logging output, set "Logging:LogLevel:Microsoft:Error" in the `appsettings.Development.json` file:

JSON

```
{
  "MyKey": "MyKey from appsettings.Development.json",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "System": "Debug",
      "Microsoft": "Error"
    }
  }
}
```

## Resolve a service at app start up

The following code shows how to resolve a scoped service for a limited duration when the app starts:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IMyDependency, MyDependency>();

var app = builder.Build();
```

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

    var myDependency = services.GetRequiredService<IMyDependency>();
    myDependency.WriteMessage("Call services from main");
}

app.MapGet("/", () => "Hello World!");

app.Run();
```

## Scope validation

See [Constructor injection behavior](#) in [Dependency injection in .NET](#)

For more information, see [Scope validation](#).

## Request Services

Services and their dependencies within an ASP.NET Core request are exposed through [HttpContext.RequestServices](#).

The framework creates a scope per request, and `RequestServices` exposes the scoped service provider. All scoped services are valid for as long as the request is active.

### ⓘ Note

Prefer requesting dependencies as constructor parameters over resolving services from `RequestServices`. Requesting dependencies as constructor parameters yields classes that are easier to test.

## Design services for dependency injection

When designing services for dependency injection:

- Avoid stateful, static classes and members. Avoid creating global state by designing apps to use singleton services instead.
- Avoid direct instantiation of dependent classes within services. Direct instantiation couples the code to a particular implementation.
- Make services small, well-factored, and easily tested.

If a class has a lot of injected dependencies, it might be a sign that the class has too many responsibilities and violates the [Single Responsibility Principle \(SRP\)](#). Attempt to refactor the class by moving some of its responsibilities into new classes. Keep in mind that Razor Pages page model classes and MVC controller classes should focus on UI concerns.

## Disposal of services

The container calls [Dispose](#) for the [IDisposable](#) types it creates. Services resolved from the container should never be disposed by the developer. If a type or factory is registered as a singleton, the container disposes the singleton automatically.

In the following example, the services are created by the service container and disposed automatically: dependency-

injection\samples\6.x\DIsample2\DIsample2\Services\Service1.cs

C#

```
public class Service1 : IDisposable
{
    private bool _disposed;

    public void Write(string message)
    {
        Console.WriteLine($"Service1: {message}");
    }

    public void Dispose()
    {
        if (_disposed)
            return;

        Console.WriteLine("Service1.Dispose");
        _disposed = true;
    }
}

public class Service2 : IDisposable
{
    private bool _disposed;

    public void Write(string message)
    {
        Console.WriteLine($"Service2: {message}");
    }

    public void Dispose()
    {
        if (_disposed)
            return;
    }
}
```

```

        Console.WriteLine("Service2.Dispose");
        _disposed = true;
    }
}

public interface IService3
{
    public void Write(string message);
}

public class Service3 : IService3, IDisposable
{
    private bool _disposed;

    public Service3(string myKey)
    {
        MyKey = myKey;
    }

    public string MyKey { get; }

    public void Write(string message)
    {
        Console.WriteLine($"Service3: {message}, MyKey = {MyKey}");
    }

    public void Dispose()
    {
        if (_disposed)
            return;

        Console.WriteLine("Service3.Dispose");
        _disposed = true;
    }
}

```

C#

```

using DIsample2.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddScoped<Service1>();
builder.Services.AddSingleton<Service2>();

var myKey = builder.Configuration["MyKey"];
builder.Services.AddSingleton<IService3>(sp => new Service3(myKey));

var app = builder.Build();

```

C#

```
public class IndexModel : PageModel
{
    private readonly Service1 _service1;
    private readonly Service2 _service2;
    private readonly IService3 _service3;

    public IndexModel(Service1 service1, Service2 service2, IService3
service3)
    {
        _service1 = service1;
        _service2 = service2;
        _service3 = service3;
    }

    public void OnGet()
    {
        _service1.Write("IndexModel.OnGet");
        _service2.Write("IndexModel.OnGet");
        _service3.Write("IndexModel.OnGet");
    }
}
```

The debug console shows the following output after each refresh of the Index page:

Console

```
Service1: IndexModel.OnGet
Service2: IndexModel.OnGet
Service3: IndexModel.OnGet, MyKey = MyKey from appsettings.Development.json
Service1.Dispose
```

## Services not created by the service container

Consider the following code:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddSingleton(new Service1());
builder.Services.AddSingleton(new Service2());
```

In the preceding code:



- The service instances aren't created by the service container.
- The framework doesn't dispose of the services automatically.
- The developer is responsible for disposing the services.

## IDisposable guidance for Transient and shared instances

See [IDisposable guidance for Transient and shared instance](#) in [Dependency injection in .NET](#)

## Default service container replacement

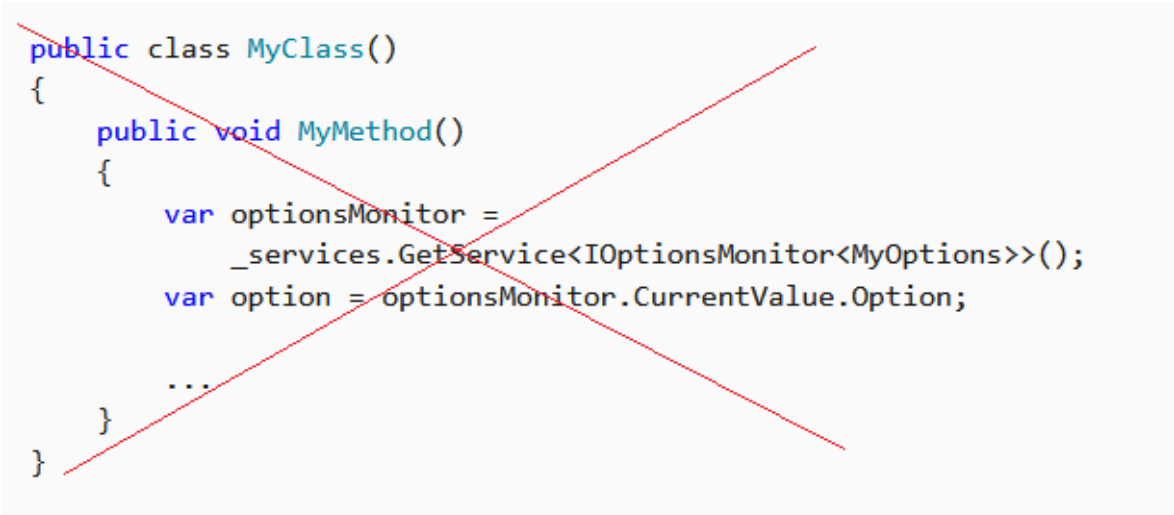
See [Default service container replacement](#) in [Dependency injection in .NET](#)

## Recommendations

See [Recommendations](#) in [Dependency injection in .NET](#)

- Avoid using the *service locator pattern*. For example, don't invoke [GetService](#) to obtain a service instance when you can use DI instead:

Incorrect:



```
public class MyClass()
{
    public void MyMethod()
    {
        var optionsMonitor =
            _services.GetService<IOptionsMonitor<MyOptions>>();
        var option = optionsMonitor.CurrentValue.Option;

        ...
    }
}
```

Correct:

```
C#

public class MyClass
{
    private readonly IOptionsMonitor<MyOptions> _optionsMonitor;

    public MyClass(IOptionsMonitor<MyOptions> optionsMonitor)
    {
```

```

        _optionsMonitor = optionsMonitor;
    }

    public void MyMethod()
    {
        var option = _optionsMonitor.CurrentValue.Option;

        ...
    }
}

```

- Another service locator variation to avoid is injecting a factory that resolves dependencies at runtime. Both of these practices mix [Inversion of Control](#) strategies.
- Avoid static access to `HttpContext` (for example, [IHttpContextAccessor.HttpContext](#)).

DI is an *alternative* to static/global object access patterns. You may not be able to realize the benefits of DI if you mix it with static object access.

## Recommended patterns for multi-tenancy in DI

[Orchard Core](#) [↗](#) is an application framework for building modular, multi-tenant applications on ASP.NET Core. For more information, see the [Orchard Core Documentation](#) [↗](#).

See the [Orchard Core samples](#) [↗](#) for examples of how to build modular and multi-tenant apps using just the Orchard Core Framework without any of its CMS-specific features.

## Framework-provided services

`Program.cs` registers services that the app uses, including platform features, such as Entity Framework Core and ASP.NET Core MVC. Initially, the `IServiceCollection` provided to `Program.cs` has services defined by the framework depending on [how the host was configured](#). For apps based on the ASP.NET Core templates, the framework registers more than 250 services.

The following table lists a small sample of these framework-registered services:

[↗](#) Expand table

Service Type	Lifetime
<a href="#">Microsoft.AspNetCore.Hosting.Builder.IApplicationBuilderFactory</a>	Transient
<a href="#">IHostApplicationLifetime</a>	Singleton
<a href="#">IWebHostEnvironment</a>	Singleton
<a href="#">Microsoft.AspNetCore.Hosting.IStartup</a>	Singleton
<a href="#">Microsoft.AspNetCore.Hosting.IStartupFilter</a>	Transient
<a href="#">Microsoft.AspNetCore.Hosting.Server.IServer</a>	Singleton
<a href="#">Microsoft.AspNetCore.Http.IHttpContextFactory</a>	Transient
<a href="#">Microsoft.Extensions.Logging.ILogger&lt;TCategoryName&gt;</a>	Singleton
<a href="#">Microsoft.Extensions.Logging.ILoggerFactory</a>	Singleton
<a href="#">Microsoft.Extensions.ObjectPool.ObjectPoolProvider</a>	Singleton
<a href="#">Microsoft.Extensions.Options.IConfigureOptions&lt;TOptions&gt;</a>	Transient
<a href="#">Microsoft.Extensions.Options.IOptions&lt;TOptions&gt;</a>	Singleton
<a href="#">System.Diagnostics.DiagnosticSource</a>	Singleton
<a href="#">System.Diagnostics.DiagnosticListener</a>	Singleton

## Additional resources

- [Dependency injection into views in ASP.NET Core](#)
- [Dependency injection into controllers in ASP.NET Core](#)
- [Dependency injection in requirement handlers in ASP.NET Core](#)
- [ASP.NET Core Blazor dependency injection](#)
- [NDC Conference Patterns for DI app development](#) ↗
- [App startup in ASP.NET Core](#)
- [Factory-based middleware activation in ASP.NET Core](#)
- [Understand dependency injection basics in .NET](#)
- [Dependency injection guidelines](#)
- [Tutorial: Use dependency injection in .NET](#)
- [.NET dependency injection](#)
- [ASP.NET CORE DEPENDENCY INJECTION: WHAT IS THE ISERVICECOLLECTION?](#) ↗
- [Four ways to dispose IDisposables in ASP.NET Core](#) ↗
- [Writing Clean Code in ASP.NET Core with Dependency Injection \(MSDN\)](#)
- [Explicit Dependencies Principle](#)

- [Inversion of Control Containers and the Dependency Injection Pattern \(Martin Fowler\)](#) ↗
- [How to register a service with multiple interfaces in ASP.NET Core DI](#) ↗

# ASP.NET Core support for Native AOT

Article • 09/18/2024

By [Mitch Denny](#) 

ASP.NET Core 8.0 introduces support for [.NET native ahead-of-time \(AOT\)](#).

For Blazor WebAssembly Native AOT guidance, which adds to or supersedes the guidance in this article, see [ASP.NET Core Blazor WebAssembly build tools and ahead-of-time \(AOT\) compilation](#).

## Why use Native AOT with ASP.NET Core

Publishing and deploying a Native AOT app provides the following benefits:

- **Minimized disk footprint:** When publishing using Native AOT, a single executable is produced containing just the code from external dependencies that is needed to support the program. Reduced executable size can lead to:
  - Smaller container images, for example in containerized deployment scenarios.
  - Reduced deployment time from smaller images.
- **Reduced startup time:** Native AOT applications can show reduced start-up times, which means
  - The app is ready to service requests quicker.
  - Improved deployment where container orchestrators need to manage transition from one version of the app to another.
- **Reduced memory demand:** Native AOT apps can have reduced memory demands, depending on the work done by the app. Reduced memory consumption can lead to greater deployment density and improved scalability.

The template app was run in our benchmarking lab to compare performance of an AOT published app, a trimmed runtime app, and an untrimmed runtime app. The following chart shows the results of the benchmarking:



The preceding chart shows that Native AOT has lower app size, memory usage, and startup time.

## ASP.NET Core and Native AOT compatibility

Not all features in ASP.NET Core are currently compatible with Native AOT. The following table summarizes ASP.NET Core feature compatibility with Native AOT:

[Expand table](#)

Feature	Fully Supported	Partially Supported	Not Supported
gRPC	✓		
Minimal APIs		✓	
MVC			✗
Blazor Server			✗
SignalR		✓	
JWT Authentication	✓		
Other Authentication			✗
CORS	✓		
HealthChecks	✓		

Feature	Fully Supported	Partially Supported	Not Supported
HttpLogging	✓		
Localization	✓		
OutputCaching	✓		
RateLimiting	✓		
RequestDecompression	✓		
ResponseCaching	✓		
ResponseCompression	✓		
Rewrite	✓		
Session			✗
Spa			✗
StaticFiles	✓		
WebSockets	✓		

For more information on limitations, see:

- [Limitations of Native AOT deployment](#)
- [Introduction to AOT warnings](#)
- [Known trimming incompatibilities](#)
- [Introduction to trim warnings](#)
- [GitHub issue dotnet/core #8288](#) ↗

It's important to test an app thoroughly when moving to a Native AOT deployment model. The AOT deployed app should be tested to verify functionality hasn't changed from the untrimmed and JIT-compiled app. When building the app, review and correct AOT warnings. An app that issues [AOT warnings](#) during publishing may not work correctly. If no AOT warnings are issued at publish time, the published AOT app should work the same as the untrimmed and JIT-compiled app.

## Native AOT publishing

Native AOT is enabled with the `PublishAot` MSBuild property. The following example shows how to enable Native AOT in a project file:

```
XML
```

```
<PropertyGroup>
  <PublishAot>true</PublishAot>
</PropertyGroup>
```

This setting enables Native AOT compilation during publish and enables dynamic code usage analysis during build and editing. A project that uses Native AOT publishing uses JIT compilation when running locally. An AOT app has the following differences from a JIT-compiled app:

- Features that aren't compatible with Native AOT are disabled and throw exceptions at run time.
- A source analyzer is enabled to highlight code that isn't compatible with Native AOT. At publish time, the entire app, including NuGet packages, are analyzed for compatibility again.

Native AOT analysis includes all of the app's code and the libraries the app depends on. Review Native AOT warnings and take corrective steps. It's a good idea to publish apps frequently to discover issues early in the development lifecycle.

In .NET 8, Native AOT is supported by the following ASP.NET Core app types:

- minimal APIs - For more information, see the [The Web API \(Native AOT\) template](#) section later in this article.
- gRPC - For more information, see [gRPC and Native AOT](#).
- Worker services - For more information, see [AOT in Worker Service templates](#).

## The Web API (Native AOT) template

The **ASP.NET Core Web API (Native AOT)** template (short name `webapiaot`) creates a project with AOT enabled. The template differs from the **Web API** project template in the following ways:

- Uses minimal APIs only, as MVC isn't yet compatible with Native AOT.
- Uses the `CreateSlimBuilder()` API to ensure only the essential features are enabled by default, minimizing the app's deployed size.
- Is configured to listen on HTTP only, as HTTPS traffic is commonly handled by an ingress service in cloud-native deployments.
- Doesn't include a launch profile for running under IIS or IIS Express.
- Creates an [.http file](#) configured with sample HTTP requests that can be sent to the app's endpoints.
- Includes a sample `Todo` API instead of the weather forecast sample.
- Adds `PublishAot` to the project file, as shown [earlier in this article](#).



- Enables the [JSON serializer source generators](#). The source generator is used to generate serialization code at build time, which is required for Native AOT compilation.

## Changes to support source generation

The following example shows the code added to the `Program.cs` file to support JSON serialization source generation:

```
diff

using MyFirstAotWebApi;
+using System.Text.Json.Serialization;

-var builder = WebApplication.CreateBuilder();
+var builder = WebApplication.CreateSlimBuilder(args);

+builder.Services.ConfigureHttpJsonOptions(options =>
+{
+ options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
+});

var app = builder.Build();

var sampleTodos = TodoGenerator.GenerateTodos().ToArray();

var todosApi = app.MapGroup("/todos");
todosApi.MapGet("/", () => sampleTodos);
todosApi.MapGet("/{id}", (int id) =>
    sampleTodos.FirstOrDefault(a => a.Id == id) is { } todo
        ? Results.Ok(todo)
        : Results.NotFound());

app.Run();

+[JsonSerializable(typeof(Todo[]))]
+internal partial class AppJsonSerializerContext : JsonSerializerContext
+{
+
+}
+}
```

Without this added code, `System.Text.Json` uses reflection to serialize and deserialize JSON. Reflection isn't supported in Native AOT.

For more information, see:

- [Combine source generators](#)
- [TypeInfoResolverChain](#)

## Changes to `launchSettings.json`

The `launchSettings.json` file created by the **Web API (Native AOT)** template has the `iisSettings` section and `IIS Express` profile removed:

diff

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
-  "iisSettings": {
-    "windowsAuthentication": false,
-    "anonymousAuthentication": true,
-    "iisExpress": {
-      "applicationUrl": "http://localhost:11152",
-      "sslPort": 0
-    }
-  },
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "launchUrl": "todos",
      "applicationUrl": "http://localhost:5102",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
-    "IIS Express": {
-      "commandName": "IISExpress",
-      "launchBrowser": true,
-      "launchUrl": "todos",
-      "environmentVariables": {
-        "ASPNETCORE_ENVIRONMENT": "Development"
-      }
-    }
  }
}
```

## The `CreateSlimBuilder` method

The template uses the `CreateSlimBuilder()` method instead of the `CreateBuilder()` method.

C#

```
using System.Text.Json.Serialization;
using MyFirstAotWebApi;

var builder = WebApplication.CreateSlimBuilder(args);
```

```

builder.Logging.AddConsole();

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

var sampleTodos = TodoGenerator.GenerateTodos().ToArray();

var todosApi = app.MapGroup("/todos");
todosApi.MapGet("/", () => sampleTodos);
todosApi.MapGet("/{id}", (int id) =>
    sampleTodos.FirstOrDefault(a => a.Id == id) is { } todo
        ? Results.Ok(todo)
        : Results.NotFound());

app.Run();

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

```

The `CreateSlimBuilder` method initializes the `WebApplicationBuilder` with the minimum ASP.NET Core features necessary to run an app.

As noted earlier, the `CreateSlimBuilder` method doesn't include support for HTTPS or HTTP/3. These protocols typically aren't required for apps that run behind a TLS termination proxy. For example, see [TLS termination and end to end TLS with Application Gateway](#). HTTPS can be enabled by calling `builder.WebHost.UseKestrelHttpsConfiguration` [↗](#) HTTP/3 can be enabled by calling `builder.WebHost.UseQuic`.

## CreateSlimBuilder vs CreateBuilder

The `CreateSlimBuilder` method doesn't support the following features that are supported by the `CreateBuilder` method:

- [Hosting startup assemblies](#)
- [UseStartup](#)
- The following logging providers:
  - [Windows EventLog](#)
  - [Debug](#)

- [Event Source](#)
- Web hosting features:
  - [UseStaticWebAssets](#)
  - [IIS Integration](#)
- Kestrel configuration
  - [HTTPS endpoints in Kestrel](#)
  - [Quic \(HTTP/3\)](#)
- [Regex and alpha constraints used in routing](#) [↗](#)

The `CreateSlimBuilder` method includes the following features needed for an efficient development experience:

- JSON file configuration for `appsettings.json` and `appsettings.{EnvironmentName}.json`.
- User secrets configuration.
- Console logging.
- Logging configuration.

For a builder that omits the preceding features, see [The CreateEmptyBuilder method](#).

Including minimal features has benefits for trimming as well as AOT. For more information, see [Trim self-contained deployments and executables](#).

For more detailed information, see [Comparing WebApplication.CreateBuilder to CreateSlimBuilder](#) [↗](#)

## Source generators

Because unused code is trimmed during publishing for Native AOT, the app can't use unbounded reflection at runtime. [Source generators](#) are used to produce code that avoids the need for reflection. In some cases, source generators produce code optimized for AOT even when a generator isn't required.

To view the source code that is generated, add the [EmitCompilerGeneratedFiles](#) property to an app's `.csproj` file, as shown in the following example:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <!-- Other properties omitted for brevity -->
    <EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>
  </PropertyGroup>
```

```
</Project>
```

Run the `dotnet build` command to see the generated code. The output includes an `obj/Debug/net8.0/generated/` directory that contains all the generated files for the project.

The `dotnet publish` command also compiles the source files and generates files that are compiled. In addition, `dotnet publish` passes the generated assemblies to a native IL compiler. The IL compiler produces the native executable. The native executable contains the native machine code.

## Libraries and Native AOT

Many of the popular libraries used in ASP.NET Core projects currently have some compatibility issues when used in a project targeting Native AOT, such as:

- Use of reflection to inspect and discover types.
- Conditionally loading libraries at runtime.
- Generating code on the fly to implement functionality.

Libraries using these dynamic features need to be updated in order to work with Native AOT. They can be updated using tools like Roslyn source generators.

Library authors hoping to support Native AOT are encouraged to:

- Read about [Native AOT compatibility requirements](#).
- [Prepare the library for trimming](#).

## Minimal APIs and JSON payloads

The Minimal API framework is optimized for receiving and returning JSON payloads using [System.Text.Json](#). `System.Text.Json`:

- Imposes compatibility requirements for JSON and Native AOT.
- Requires the use of the [System.Text.Json source generator](#).

All types that are transmitted as part of the HTTP body or returned from request delegates in Minimal APIs apps must be configured on a [JsonSerializerContext](#) that is registered via ASP.NET Core's dependency injection:

```
C#
```

```

using System.Text.Json.Serialization;
using MyFirstAotWebApi;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Logging.AddConsole();

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

var sampleTodos = TodoGenerator.GenerateTodos().ToArray();

var todosApi = app.MapGroup("/todos");
todosApi.MapGet("/", () => sampleTodos);
todosApi.MapGet("/{id}", (int id) =>
    sampleTodos.FirstOrDefault(a => a.Id == id) is { } todo
        ? Results.Ok(todo)
        : Results.NotFound());

app.Run();

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

```

In the preceding highlighted code:

- The JSON serializer context is registered with the [DI container](#). For more information, see:
  - [Combine source generators](#)
  - [TypeInfoResolverChain](#)
- The custom `JsonSerializerContext` is annotated with the [\[JsonSerializable\]](#) attribute to enable source generated JSON serializer code for the `ToDo` type.

A parameter on the delegate that isn't bound to the body and does **not** need to be serializable. For example, a query string parameter that is a rich object type and implements `IParsable<T>`.

C#

```

public class Todo
{
    public int Id { get; set; }
}

```

```

    public string? Title { get; set; }
    public DateOnly? DueBy { get; set; }
    public bool IsComplete { get; set; }
}

static class TodoGenerator
{
    private static readonly (string[] Prefixes, string[] Suffixes)[] _parts
    = new[]
    {
        (new[] { "Walk the", "Feed the" }, new[] { "dog", "cat", "goat"
    })),
        (new[] { "Do the", "Put away the" }, new[] { "groceries",
"dishes", "laundry" })),
        (new[] { "Clean the" }, new[] { "bathroom", "pool", "blinds",
"car" })
    };
    // Remaining code omitted for brevity.

```

## Known issues

See [this GitHub issue](#) to report or review issues with Native AOT support in ASP.NET Core.

## See also

- [Tutorial: Publish an ASP.NET Core app using Native AOT](#)
- [Native AOT deployment](#)
- [Optimize AOT deployments](#)
- [Configuration-binding source generator](#)
- [Using the configuration binder source generator](#)
- [The minimal API AOT compilation template](#)
- [Comparing WebApplication.CreateBuilder to CreateSlimBuilder](#)
- [Exploring the new minimal API source generator](#)
- [Replacing method calls with Interceptors](#)
- [Behind \[LogProperties\] and the new telemetry logging source generator](#)

 **Note:** The author created this article with assistance from AI. [Learn more](#)

# Tutorial: Publish an ASP.NET Core app using Native AOT

Article • 07/26/2024

ASP.NET Core 8.0 introduces support for [.NET native ahead-of-time \(AOT\)](#).

## ⓘ Note

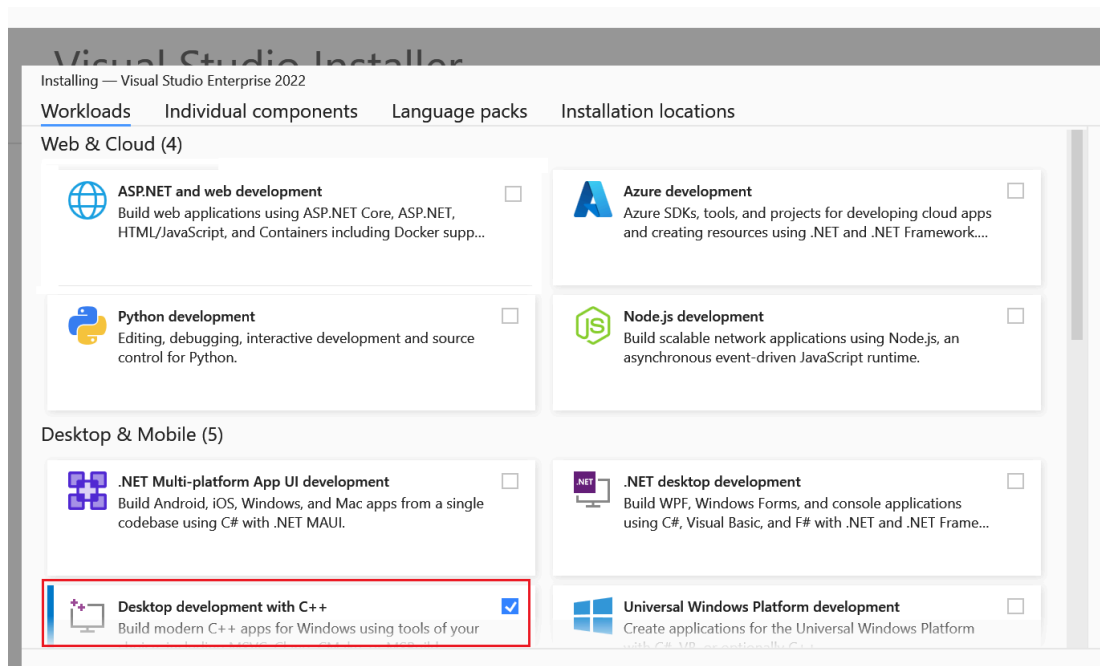
- The Native AOT feature is currently in preview.
- In .NET 8, not all ASP.NET Core features are compatible with Native AOT.
- Tabs are provided for the [.NET CLI](#) and [Visual Studio](#) instructions:
  - Visual Studio is a prerequisite even if the CLI tab is selected.
  - The CLI must be used to publish even if the Visual Studio tab is selected.

## Prerequisites

### .NET CLI

- [.NET 8.0 SDK](#)
- On Linux, see [Prerequisites for Native AOT deployment](#).
- [Visual Studio 2022 Preview](#) with the **Desktop development with C++** workload installed.





### ⓘ Note

Visual Studio 2022 Preview is required because Native AOT requires [link.exe](#) and the Visual C++ static runtime libraries. There are no plans to support Native AOT *without* Visual Studio.

## Create a web app with Native AOT

Create an ASP.NET Core API app that is configured to work with Native AOT:

### .NET CLI

Run the following commands:

.NET CLI

```
dotnet new webapiaot -o MyFirstAotWebApi && cd MyFirstAotWebApi
```

Output similar to the following example is displayed:

### Output

The template "ASP.NET Core Web API (Native AOT)" was created successfully.

Processing post-creation actions...

Restoring C:\Code\Demos\MyFirstAotWebApi\MyFirstAotWebApi.csproj:

```
Determining projects to restore...
Restored C:\Code\Demos\MyFirstAotWebApi\MyFirstAotWebApi.csproj (in
302 ms).
Restore succeeded.
```

## Publish the Native AOT app

Verify the app can be published using Native AOT:

.NET CLI

.NET CLI

```
dotnet publish
```

The `dotnet publish` command:

- Compiles the source files.
- Generates source code files that are compiled.
- Passes generated assemblies to a native IL compiler. The IL compiler produces the native executable. The native executable contains the native machine code.

Output similar to the following example is displayed:

### Output

```
MSBuild version 17.<version> for .NET
Determining projects to restore...
Restored C:\Code\Demos\MyFirstAotWebApi\MyFirstAotWebApi.csproj (in 241
ms).
C:\Code\dotnet\aspnetcore\dotnet\sdk\8.0.
<version>\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.RuntimeIde
ntifierInference.targets(287,5): message NETSDK1057: You are using a preview
version of .NET. See: https://aka.ms/dotne
t-support-policy [C:\Code\Demos\MyFirstAotWebApi\MyFirstAotWebApi.csproj]
MyFirstAotWebApi -> C:\Code\Demos\MyFirstAotWebApi\bin\Release\net8.0\win-
x64\MyFirstAotWebApi.dll
Generating native code
MyFirstAotWebApi -> C:\Code\Demos\MyFirstAotWebApi\bin\Release\net8.0\win-
x64\publish\
```

The output may differ from the preceding example depending on the version of .NET 8 used, directory used, and other factors.

Review the contents of the output directory:

```
dir bin\Release\net8.0\win-x64\publish
```

Output similar to the following example is displayed:

Output

```
Directory: C:\Code\Demos\MyFirstAotWebApi\bin\Release\net8.0\win-x64\publish
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	30/03/2023 1:41 PM	9480704	MyFirstAotWebApi.exe
-a---	30/03/2023 1:41 PM	43044864	MyFirstAotWebApi.pdb

The executable is self-contained and doesn't require a .NET runtime to run. When launched, it behaves the same as the app run in the development environment. Run the AOT app:

```
.\bin\Release\net8.0\win-x64\publish\MyFirstAotWebApi.exe
```

Output similar to the following example is displayed:

Output

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Code\Demos\MyFirstAotWebApi
```

## Libraries and Native AOT

Many of the popular libraries used in ASP.NET Core projects currently have some compatibility issues when used in a project targeting Native AOT, such as:

- Use of reflection to inspect and discover types.

- Conditionally loading libraries at runtime.
- Generating code on the fly to implement functionality.

Libraries using these dynamic features need to be updated in order to work with Native AOT. They can be updated using tools like Roslyn source generators.

Library authors hoping to support Native AOT are encouraged to:

- Read about [Native AOT compatibility requirements](#).
- [Prepare the library for trimming](#).

## See also

- [ASP.NET Core support for Native AOT](#)
- [Native AOT deployment](#)
- [Using the configuration binder source generator](#) ↗
- [The minimal API AOT compilation template](#) ↗
- [Comparing WebApplication.CreateBuilder to CreateSlimBuilder](#) ↗
- [Exploring the new minimal API source generator](#) ↗
- [Replacing method calls with Interceptors](#) ↗
- [Configuration-binding source generator](#)

 **Note:** The author created this article with assistance from AI. [Learn more](#)

# Turn Map methods into request delegates with the ASP.NET Core Request Delegate Generator

Article • 07/26/2024

The ASP.NET Core Request Delegate Generator (RDG) is a compile-time source generator that compiles route handlers provided to a minimal API to request delegates that can be processed by ASP.NET Core's routing infrastructure. The RDG is implicitly enabled when applications are published with AoT enabled or when [trimming is enabled](#). The RDG generates trim and native AoT-friendly code.

## ⓘ Note

- The Native AOT feature is currently in preview.
- In .NET 8, not all ASP.NET Core features are compatible with Native AOT.

The RDG:

- Is a [source generator](#).
- Turns each `Map` method into a [RequestDelegate](#) associated with the specific route. `Map` methods include the methods in the [EndpointRouteBuilderExtensions](#), such as [MapGet](#), [MapPost](#), [MapPatch](#), [MapPut](#), and [MapDelete](#).

When publishing and Native AOT is **not** enabled:

- `Map` methods associated with a specific route are compiled in memory into a request delegate when the app starts, not when the app is built.
- The request delegates are generated at runtime.

When publishing with Native AOT enabled:

- `Map` methods associated with a specific route are compiled when the app is built. The RDG creates the request delegate for the route and the request delegate is compiled into the app's native image.
- Eliminates the need to generate the request delegate at runtime.
- Ensures:
  - The types used in the app's APIs are rooted in the app code in a way that is statically analyzable by the Native AOT tool-chain.
  - The required code isn't trimmed away.

The RDG:

- Is enabled automatically in projects when publishing with Native AOT is enabled or when trimming is enabled.
- Can be manually enabled even when not using Native AOT by setting `<EnableRequestDelegateGenerator>true</EnableRequestDelegateGenerator>` in the project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <EnableRequestDelegateGenerator>true</EnableRequestDelegateGenerator>
  </PropertyGroup>

</Project>
```

Manually enabling RDG can be useful for:

- Evaluating a project's compatibility with Native AOT.
- Reducing the app's startup time by pregenerating the request delegates.

Minimal APIs are optimized for using [System.Text.Json](#), which requires using the [System.Text.Json source generator](#). All types accepted as parameters to or returned from request delegates in Minimal APIs must be configured on a [JsonSerializerContext](#) that's registered via ASP.NET Core's dependency injection:

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(
        0, AppJsonSerializerContext.Default);
});

var app = builder.Build();

var sampleTodos = new Todo[] {
    new(1, "Walk the dog"),
    new(2, "Do the dishes", DateOnly.FromDateTime(DateTime.Now)),
    new(3, "Do the laundry",
```

```

DateOnly.FromDateTime(DateTime.Now.AddDays(1))),
    new(4, "Clean the bathroom"),
    new(5, "Clean the car", DateOnly.FromDateTime(DateTime.Now.AddDays(2)))
};

var todosApi = app.MapGroup("/todos");
todosApi.MapGet("/", () => sampleTodos);
todosApi.MapGet("/{id}", (int id) =>
    sampleTodos.FirstOrDefault(a => a.Id == id) is { } todo
    ? Results.Ok(todo)
    : Results.NotFound());

app.Run();

public record Todo(int Id, string? Title, DateOnly? DueBy = null, bool
IsComplete = false);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

```

## Diagnostics for unsupported RDG scenarios

When the app is built, the RDG emits diagnostics for scenarios that aren't supported by Native AOT. The diagnostics are emitted as warnings and don't prevent the app from building. For the list of diagnostics, see [ASP.NET Core Request Delegate Generator diagnostics](#).

 **Note:** The author created this article with assistance from AI. [Learn more](#)

# ASP.NET Core Request Delegate Generator (RDG) diagnostics

Article • 07/26/2024

The ASP.NET Core Request Delegate Generator (RDG) is a tool that generates request delegates for ASP.NET Core apps. The RDG is used by the native ahead-of-time (AOT) compiler to generate request delegates for the app's `Map` methods.

## ⓘ Note

- The Native AOT feature is currently in preview.
- In .NET 8, not all ASP.NET Core features are compatible with Native AOT.

The following list contains the [RDG diagnostics](#) for ASP.NET Core:

- [RDG002: Unable to resolve endpoint handler](#)
- [RDG004: Unable to resolve anonymous type](#)
- [RDG005: Invalid abstract type](#)
- [RDG006: Invalid constructor parameters](#)
- [RDG007: No valid constructor found](#)
- [RDG008: Multiple public constructors](#)
- [RDG009: Invalid nested AsParameters](#)
- [RDG010: InvalidAsParameters Nullable](#)
- [RDG011: Type parameters not supported](#)
- [RDG012: Unable to resolve inaccessible type](#)
- [RDG013: Invalid source attributes](#)

ⓘ **Note:** The author created this article with assistance from AI. [Learn more](#)



# RDG002: Unable to resolve endpoint handler

Article • 07/26/2024

[Expand table](#)

	Value
Rule ID	RDG002
Fix is breaking or non-breaking	Non-breaking

## Cause

This diagnostic is emitted by the Request Delegate Generator when an endpoint contains a route handler that can't be statically analyzed.

## Rule description

The Request Delegate Generator runs at compile-time and needs to be able to statically analyze route handlers in an app. The current implementation only supports route handlers that are provided as a lambda expression, method group references, or references to read-only fields or variables.

The following code generates the RDG002 warning because the route handler is provided as a reference to a method:

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();

var del = Wrapper.GetTodos;
app.MapGet("/v1/todos", del);
```

```

app.Run();

record Todo(int Id, string Task);
[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

class Wrapper
{
    public static Func<IResult> GetTodos = () =>
        Results.Ok(new Todo(1, "Write test fix"));
}

```

## How to fix violations

Declare the route handler using supported syntax, such as an inline lambda:

```

C#

using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/v1/todos", () => Results.Ok(new Todo(1, "Write tests")));

app.Run();

record Todo(int Id, string Task);
[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}


```

## When to suppress warnings

This warning can be safely suppressed. When suppressed, the framework falls back to generating the request delegate at runtime.

# RDG004: Unable to resolve anonymous type

Article • 07/26/2024

 Expand table

	Value
Rule ID	RDG004
Fix is breaking or non-breaking	Non-breaking

## Cause

This diagnostic is emitted by the [Request Delegate Generator](#) when an endpoint contains a route handler with an anonymous return type.

## Rule description

The Request Delegate Generator runs at compile-time and needs to be able to statically analyze route handlers in an app. [Anonymous types](#) are generated with a type name only known to the compiler and aren't statically analyzable. The following endpoint produces the diagnostic.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/v1/todos", () => new { Id = 1, Task = "Write tests" });

app.Run();

record Todo(int Id, string Task);
[JsonSerializable(typeof(Todo[]))]
```

```
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

## How to fix violations

Declare the route handler with a concrete type as the return type.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/v1/todos", () => new Todo(1, "Write tests fix"));

app.Run();

record Todo(int Id, string Task);
[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

## When to suppress warnings

This warning can be safely suppressed. When suppressed, the framework falls back to generating the request delegate at runtime.

 **Note:** The author created this article with assistance from AI. [Learn more](#)

# RDG005: Invalid abstract type

Article • 07/26/2024

 Expand table

	Value
Rule ID	RDG005
Fix is breaking or non-breaking	Non-breaking

## Cause

This diagnostic is emitted by the [Request Delegate Generator](#) when an endpoint contains a route handler with a parameter annotated with the [\[AsParameters\]](#) attribute that is an abstract type.

## Rule description

The implementation of surrogate binding via the [\[AsParameters\]](#) attribute in minimal APIs only supports types with concrete implementations. Using a parameter with an [abstract](#) type as in the following sample produces the diagnostic.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();
app.MapPut("/v1/todos/{id}",
    ([AsParameters] TodoRequest todoRequest) =>
    Results.Ok(todoRequest.Todo));

app.Run();
```

```

abstract class TodoRequest
{
    public int Id { get; set; }
    public Todo? Todo { get; set; }
}

record Todo(int Id, string Task);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

```

## How to fix violations

Use a concrete type for the surrogate:

C#

```

using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapPut("/v1/todos/{id}",
    ([AsParameters] TodoRequest todoRequest) =>
    Results.Ok(todoRequest.Todo));

app.Run();

class TodoRequest
{
    public int Id { get; set; }
    public Todo? Todo { get; set; }
}

record Todo(int Id, string Task);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

```

# When to suppress warnings

This warning should not be suppressed. Suppressing the warning will lead to a runtime exception associated with the same warning.

ⓘ **Note:** The author created this article with assistance from AI. [Learn more](#)

# RDG006: Invalid constructor parameters

Article • 07/26/2024

 Expand table

	Value
Rule ID	RDG006
Fix is breaking or non-breaking	Non-breaking

## Cause

This diagnostic is emitted by the [Request Delegate Generator](#) when an endpoint contains a route handler with a parameter annotated with the [\[AsParameters\]](#) attribute that contains an invalid constructor.

## Rule description

Types that are used for surrogate binding via the [\[AsParameters\]](#) attribute must contain a public parameterized constructor where all parameters to the constructor match the public properties declared on the type. The `TodoRequest` type produces this diagnostic because there is no matching constructor parameter for the `Todo` property.

```
C#

using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                                                    AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapPut("/v1/todos/{id}",
           ([AsParameters] TodoRequest todoRequest) =>
           Results.Ok(todoRequest.TODO));

app.Run();
```



```

class TodoRequest(int id, string name)
{
    public int Id { get; set; } = id;
    public Todo? Todo { get; set; }
}

record Todo(int Id, string Task);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

```

## How to fix violations

Ensure that all properties on the type have a matching parameter on the constructor.

C#

```

using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapPut("/v1/todos/{id}",
    ([AsParameters] TodoRequest todoRequest) =>
    Results.Ok(todoRequest.Todo));

app.Run();

class TodoRequest(int id, Todo? todo)
{
    public int Id { get; set; } = id;
    public Todo? Todo { get; set; } = todo;
}

record Todo(int Id, string Task);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

```

# When to suppress warnings

This warning should ***not*** be suppressed. Suppressing the warning leads to a runtime exception associated with the same warning.

# RDG007: No valid constructor found

Article • 07/26/2024

 Expand table

	Value
Rule ID	RDG007
Fix is breaking or non-breaking	Non-breaking

## Cause

This diagnostic is emitted by the [Request Delegate Generator](#) when an endpoint contains a route handler with a parameter annotated with the [\[AsParameters\]](#) attribute with no valid constructor.

## Rule description

Types that are used for surrogate binding via the `AsParameters` attribute must contain a public constructor. The `TodoRequest` type produces this diagnostic because there is no public constructor.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapPut("/v1/todos/{id}", ([AsParameters] TodoRequest request)
=> Results.Ok(request.TODO));

app.Run();

public class TodoRequest
{
    public int Id { get; set; }
    public Todo Todo { get; set; }
```

```

        private TodoRequest()
        {
        }
    }

    public record Todo(int Id, string Task);

    [JsonSerializable(typeof(Todo[]))]
    internal partial class AppJsonSerializerContext : JsonSerializerContext
    {
    }

```

## How to fix violations

Remove the non-public constructor, or add a new public constructor.

C#

```

using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapPut("/v1/todos/{id}", ([AsParameters] TodoRequest request)
=> Results.Ok(request.Todo));
app.Run();

public class TodoRequest(int Id, Todo todo)
{
    public int Id { get; set; } = Id;
    public Todo Todo { get; set; } = todo;
}

public record Todo(int Id, string Task);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}


```

# When to suppress warnings

This warning can't be safely suppressed. When suppressed, results in the `InvalidOperationException` runtime exception.

# RDG008: Multiple public constructors

Article • 07/26/2024

 Expand table

	Value
Rule ID	RDG008
Fix is breaking or non-breaking	Non-breaking

## Cause

This diagnostic is emitted by the [Request Delegate Generator](#) when an endpoint contains a route handler with a parameter annotated with the [\[AsParameters\]](#) attribute with multiple public constructors.

## Rule description

Types that are used for surrogate binding via the `AsParameters` attribute must contain a single public constructor. The `TodoRequest` type produces this diagnostic because there are multiple public constructors.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder();
var todos = new[]
{
    new Todo(1, "Write tests", DateTime.UtcNow.AddDays(2)),
    new Todo(2, "Fix tests", DateTime.UtcNow.AddDays(1))
};
builder.Services.AddSingleton(todos);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/v1/todos/{id}", ([AsParameters] TodoItemRequest request) =>
{
    return request.Todos.ToList().Find(todoItem => todoItem.Id ==
request.Id)
```

```

is Todo todo
    ? Results.Ok(todo)
    : Results.NotFound();
});

app.Run();

public class TodoItemRequest
{
    public int Id { get; set; }
    public Todo[] Todos { get; set; }

    public TodoItemRequest(int id, Todo[] todos)
    {
        Id = id;
        Todos = todos;
    }

    // Additional Constructor
    public TodoItemRequest()
    {
        Id = 1;
        Todos = [new Todo(1, "Write tests", DateTime.UtcNow.AddDays(2))];
    }
}

public class Todo
{
    public DateTime DueDate { get; }
    public int Id { get; private set; }
    public string Task { get; private set; }

    public Todo(int id, string task, DateTime dueDate)

    {
        Id = id;
        Task = task;
        DueDate = dueDate;
    }
}

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

```

## How to fix violations

Provide a single public constructor.

```

using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder();
var todos = new[]
{
    new Todo(1, "Write tests", DateTime.UtcNow.AddDays(2)),
    new Todo(2, "Fix tests", DateTime.UtcNow.AddDays(1))
};
builder.Services.AddSingleton(todos);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/v1/todos/{id}", ([AsParameters] TodoItemRequest request) =>
{
    return request.Todos.ToList().Find(todoItem => todoItem.Id ==
request.Id)
is Todo todo
    ? Results.Ok(todo)
    : Results.NotFound();
});

app.Run();

public class TodoItemRequest
{
    public int Id { get; set; }
    public Todo[] Todos { get; set; }

    public TodoItemRequest(int id, Todo[] todos)
    {
        Id = id;
        Todos = todos;
    }
}

public class Todo
{
    public DateTime DueDate { get; }
    public int Id { get; private set; }
    public string Task { get; private set; }

    public Todo(int id, string task, DateTime dueDate)
    {
        Id = id;
        Task = task;
        DueDate = dueDate;
    }
}

```




```
[JsonSerializable(typeof(Todo[]))]  
internal partial class AppJsonSerializerContext : JsonSerializerContext  
{  
}
```

## When to suppress warnings

This warning can be safely suppressed.

# RDG009: Invalid nested AsParameters

Article • 07/26/2024

 Expand table

	Value
Rule ID	RDG009
Fix is breaking or non-breaking	Non-breaking

## Cause

This diagnostic is emitted by the [Request Delegate Generator](#) when an endpoint contains invalid nested [\[AsParameters\]](#).

## Rule description

Types that are used for surrogate binding via the [\[AsParameters\]](#) attribute must not contain nested types that are also annotated with the [\[AsParameters\]](#) attribute:

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder();

var todos = new[]
{
    new Todo(1, "Write tests", DateTime.UtcNow.AddDays(2)),
    new Todo(2, "Fix tests",DateTime.UtcNow.AddDays(1))
};

builder.Services.AddSingleton(todos);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/v1/todos/{id}", ([AsParameters] TodoItemRequest request) =>
{
    return request.Todos.ToList().Find(todoItem => todoItem.Id ==
request.Id) is Todo todo
```

```

        ? Results.Ok(todo)
        : Results.NotFound();
    });

app.Run();

struct TodoItemRequest
{
    public int Id { get; set; }
    [AsParameters]
    public Todo[] Todos { get; set; }
}

internal record Todo(int Id, string Task, DateTime DueDate);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

```

## How to fix violations

Remove the nested `AsParameters` attribute:

```

C#

using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder();
var todos = new[]
{
    new Todo(1, "Write tests", DateTime.UtcNow.AddDays(2)),
    new Todo(2, "Fix tests", DateTime.UtcNow.AddDays(1))
};
builder.Services.AddSingleton(todos);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/v1/todos/{id}", ([AsParameters] TodoItemRequest request) =>
{
    return request.Todos.ToList().Find(todoItem => todoItem.Id ==
request.Id) is Todo todo
        ? Results.Ok(todo)
        : Results.NotFound();
});

```

```
app.Run();

struct TodoItemRequest
{
    public int Id { get; set; }
    // [AsParameters]
    public Todo[] Todos { get; set; }
}

internal record Todo(int Id, string Task, DateTime DueDate);


[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}
```

## When to suppress warnings

This warning can ***not*** be suppressed.

# RDG010: InvalidAsParameters Nullable

Article • 07/26/2024

 Expand table

	Value
Rule ID	RDG010
Fix is breaking or non-breaking	Non-breaking

## Cause

This diagnostic is emitted by the [Request Delegate Generator](#) when an endpoint contains a route handler with a parameter annotated with the [\[AsParameters\]](#) attribute that is marked as nullable.

## Rule description

The implementation of surrogate binding via the [\[AsParameters\]](#) attribute in minimal APIs only supports types that are not nullable.

C#

```
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/todos/{id}", ([AsParameters] TodoRequest? request)
    => Results.Ok(new Todo(request!.Id)));

app.Run();

public record TodoRequest(HttpContext HttpContext, [FromRoute] int Id);
public record Todo(int Id);
[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
```

```
}
```

## How to fix violations

Declare the parameter as non-nullable.

C#

```
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/todos/{id}", ([AsParameters] TodoRequest request)
    => Results.Ok(new Todo(request.Id)));

app.Run();

public record TodoRequest(HttpContext HttpContext, [FromRoute] int Id);
public record Todo(int Id);
[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}
```

## When to suppress warnings

This warning should **not** be suppressed. Suppressing the warning leads to a runtime exception associated with the same warning.

# RDG011: Type parameters not supported

Article • 07/26/2024

 Expand table

	Value
Rule ID	RDG011
Fix is breaking or non-breaking	Non-breaking

## Cause

This diagnostic is emitted by the Request Delegate Generator when an endpoint contains a route handler that captures a generic type.

## Rule description

Endpoints that use generic type parameters are not supported. The endpoints within `MapEndpoints` produce this diagnostic because of the generic `<T>` parameter.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();
app.MapEndpoints<Todo>();
app.Run();

public static class RouteBuilderExtensions
{
    public static IEndpointRouteBuilder MapEndpoints<T>(this
IEndpointRouteBuilder app) where T : class, new()
    {
        app.MapPost("/input", (T value) => value);
        app.MapGet("/result", () => new T());
        app.MapPost("/input-with-wrapper", (Wrapper<T> value) => value);
        app.MapGet("/async", async () =>
        {
```

```

        await Task.CompletedTask;
        return new T();
    });
    return app;
}

}

record Wrapper<T>(T Value);
class Todo
{
    public int Id { get; set; }
    public string Task { get; set; }
}
[JsonSerializable(typeof(Wrapper<Todo>[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

```

## How to fix violations

Remove the generic type from endpoints.

C#

```

using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();
app.MapTodoEndpoints();
app.Run();

public static class TodoRouteBuilderExtensions
{
    public static IEndpointRouteBuilder MapTodoEndpoints(this
IEndpointRouteBuilder app)
    {
        app.MapPost("/input", (Todo value) => value);
        app.MapGet("/result", () => new Todo(1, "Walk the dog"));
        app.MapPost("/input-with-wrapper", (Wrapper<Todo> value) => value);
        app.MapGet("/async", async () =>
        {
            await Task.CompletedTask;
            return new Todo(1, "Walk the dog");
        });
        return app;
    }
}

```



```
    }  
}  
  
record Wrapper<T>(T Value);  
record Todo(int Id, string Task);  
[JsonSerializable(typeof(Wrapper<Todo>[]))]  
internal partial class AppJsonSerializerContext : JsonSerializerContext  
{  
  
}
```

## When to suppress warnings

This warning can be safely suppressed. When suppressed, the framework will fallback to generating the request delegate at runtime.

# RDG012: Unable to resolve inaccessible type

Article • 07/26/2024

[Expand table](#)

	Value
Rule ID	RDG012
Fix is breaking or non-breaking	Non-breaking

## Cause

This diagnostic is emitted by the Request Delegate Generator when an endpoint contains a route handler with a parameter without the appropriate accessibility modifiers.

## Rule description

Endpoints that use an inaccessible type (`private` or `protected`) are not supported. The endpoints within `MapEndpoints` produce this diagnostic because of the `Todo` type has the `private` accessibility modifiers.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();
app.MapEndpoints();
app.Run();

public static class TodoRouteBuilderExtensions
{
```

```

    public static IEndpointRouteBuilder MapEndpoints(this
IEndpointRouteBuilder app)
    {
        app.MapPost("/input", (Todo value) => value);
        app.MapGet("/result", () => new Todo(1, "Walk the dog"));
        app.MapPost("/input-with-wrapper", (Wrapper<Todo> value) => value);
        app.MapGet("/async", async () =>
        {
            await Task.CompletedTask;
            return new Todo(1, "Walk the dog");
        });
        return app;
    }

```

```

    private record Todo(int Id, string Task);
}

record Wrapper<T>(T Value);
[JsonSerializable(typeof(Wrapper<TodoRouteBuilderExtensions.Todo>[ ]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

```

## How to fix violations

When applicable, set the target parameter type with a friendly accessibility.

C#

```

using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

var app = builder.Build();
app.MapEndpoints();
app.Run();

public static class TodoRouteBuilderExtensions
{

```

```

public static IEndpointRouteBuilder MapEndpoints(this
IEndpointRouteBuilder app)
{
    app.MapPost("/input", (Todo value) => value);
    app.MapGet("/result", () => new Todo(1, "Walk the dog"));
    app.MapPost("/input-with-wrapper", (Wrapper<Todo> value) => value);
    app.MapGet("/async", async () =>
    {
        await Task.CompletedTask;
        return new Todo(1, "Walk the dog");
    });
    return app;
}

```

```

public record Todo(int Id, string Task);
}

record Wrapper<T>(T Value);
[JsonSerializable(typeof(Wrapper<TodoRouteBuilderExtensions.Todo>[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

```

## When to suppress warnings

This warning can be safely suppressed. When suppressed, the framework will fallback to generating the request delegate at runtime.

# RDG013: Invalid source attributes

Article • 07/26/2024

 Expand table

	Value
Rule ID	RDG013
Fix is breaking or non-breaking	Non-breaking

## Cause

This diagnostic is emitted by the [Request Delegate Generator](#) when an endpoint contains a route handler with a parameter that contains an invalid combination of service source attributes.

## Rule description

ASP.NET Core supports resolving keyed and non-keyed services via [dependency injection](#). It's *not* feasible to resolve a service as both keyed and non-keyed. The following code produces the diagnostic and throws a run time error with the same message:

C#

```
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});
builder.Services.AddKeyedSingleton<IService, FizzService>("fizz");

var app = builder.Build();

app.MapGet("/fizz", ([FromKeyedServices("fizz")][FromServices] IService
service) =>
{
    return Results.Ok(service.Echo());
});
```

```
app.Run();
```

## How to fix violations

Resolve the target parameter as either a keyed or non-keyed service.

C#

```
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
        AppJsonSerializerContext.Default);
});

builder.Services.AddKeyedSingleton<IService, FizzService>("fizz");
builder.Services.AddKeyedSingleton<IService, BuzzService>("buzz");
builder.Services.AddSingleton<IService, FizzBuzzService>();
var app = builder.Build();

app.MapGet("/fizz", ([FromKeyedServices("fizz")] IService service) =>
{
    return Results.Ok(service.Echo());
});

app.MapGet("/buzz", ([FromKeyedServices("buzz")] IService service) =>
{
    return Results.Ok(service.Echo());
});

app.MapGet("/fizzbuzz", ([FromServices] IService service) =>
{
    return Results.Ok(service.Echo());
});

app.Run();

public interface IService
{
    string Echo();
}

public class FizzService : IService
{
    public string Echo() => "Fizz";
}

public class BuzzService : IService
```

```

{
    public string Echo() => "Buzz";
}

public class FizzBuzzService : IService
{
    public string Echo()
    {
        return "FizzBuzz";
    }
}
[JsonSerializable(typeof(string[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

```

## When to suppress warnings

This warning should ***not*** be suppressed. Suppressing the warning leads to a [NotSupportedException](#) runtime exception `The FromKeyedServicesAttribute is not supported on parameters that are also annotated with IFromServiceMetadata.`

# ASP.NET Core Middleware

Article • 06/17/2024

## Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

By [Rick Anderson](#) and [Steve Smith](#)

Middleware is software that's assembled into an app pipeline to handle requests and responses. Each component:

- Chooses whether to pass the request to the next component in the pipeline.
- Can perform work before and after the next component in the pipeline.

Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.

Request delegates are configured using [Run](#), [Map](#), and [Use](#) extension methods. An individual request delegate can be specified in-line as an anonymous method (called in-line middleware), or it can be defined in a reusable class. These reusable classes and in-line anonymous methods are *middleware*, also called *middleware components*. Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline or short-circuiting the pipeline. When a middleware short-circuits, it's called a *terminal middleware* because it prevents further middleware from processing the request.

[Migrate HTTP handlers and modules to ASP.NET Core middleware](#) explains the difference between request pipelines in ASP.NET Core and ASP.NET 4.x and provides additional middleware samples.

## Middleware code analysis

ASP.NET Core includes many compiler platform analyzers that inspect application code for quality. For more information, see [Code analysis in ASP.NET Core apps](#)