

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

03/27/2019 • 14 minutes to read •  +6

In this article

[Prerequisites](#)

[Add column sort links](#)

[Add a Search box](#)

[Add paging to Students Index](#)

[Add paging to Index method](#)

[Add paging links](#)

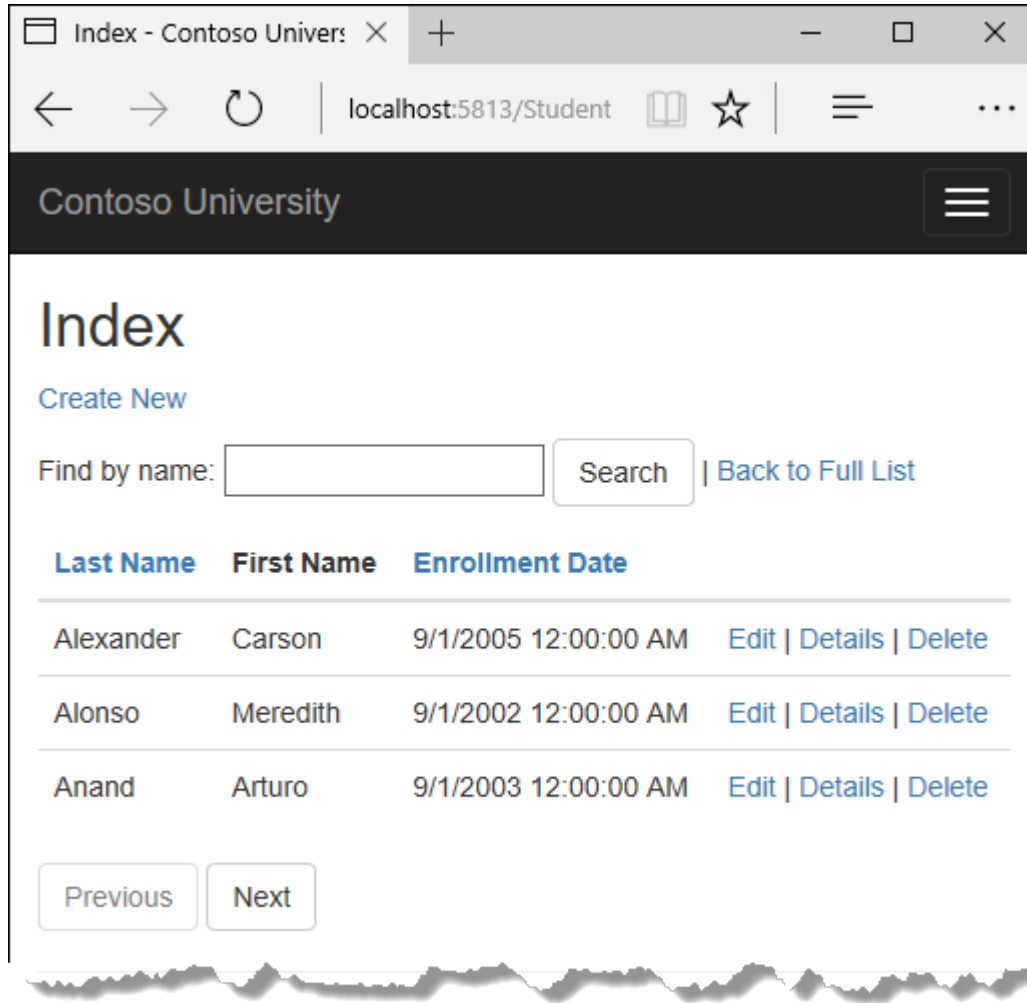
[Create an About page](#)

[Get the code](#)

[Next steps](#)

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

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



In this tutorial, you:

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

Prerequisites

- [Implement CRUD Functionality](#)

Add column sort links

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

Add sorting Functionality to the Index method

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

C#

 Copy

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

```
        break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

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

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

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

C#



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

```
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

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

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

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

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

Add column heading hyperlinks to the Student Index view

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

HTML

 Copy

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

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

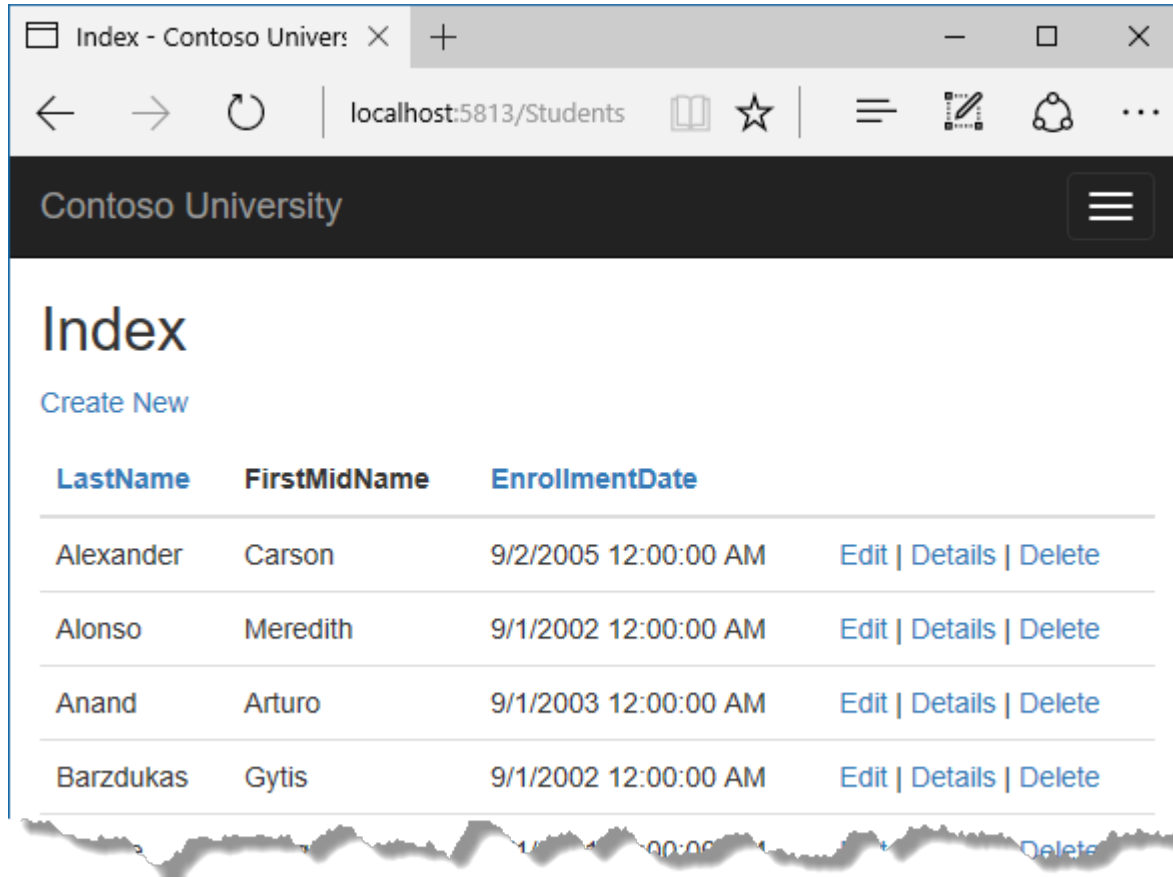
<h2>Index</h2>

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

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

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

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



Add a Search box

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

Add filtering functionality to the Index method

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

```
C#
```

Copy


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

    var students = from s in _context.Students
                   select s;

    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                                   || s.FirstMidName.Contains(searchString));
    }

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

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

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


ⓘ Note

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

For example, the .NET Framework implementation of the `Contains` method performs a case-sensitive comparison by default, but in SQL Server this is determined by the collation setting of the SQL Server instance. That setting defaults to case-insensitive. You could call the `ToUpper` method to make the test explicitly case-insensitive: *Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper()))*. That would ensure that results stay the same if you change the code later to use a repository which returns an `IEnumerable` collection instead of an `IQueryable` object. (When you call the `Contains` method on an `IEnumerable` collection, you get the .NET Framework implementation; when you call it on an `IQueryable` object, you get the database provider implementation.) However, there's a performance penalty for this solution. The `ToUpper` code would put a function in the WHERE clause of the TSQL SELECT statement. That would prevent the optimizer from using an index. Given that SQL is mostly installed as case-insensitive, it's best to avoid the `ToUpper` code until you migrate to a case-sensitive data store.

Add a Search Box to the Student Index View

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

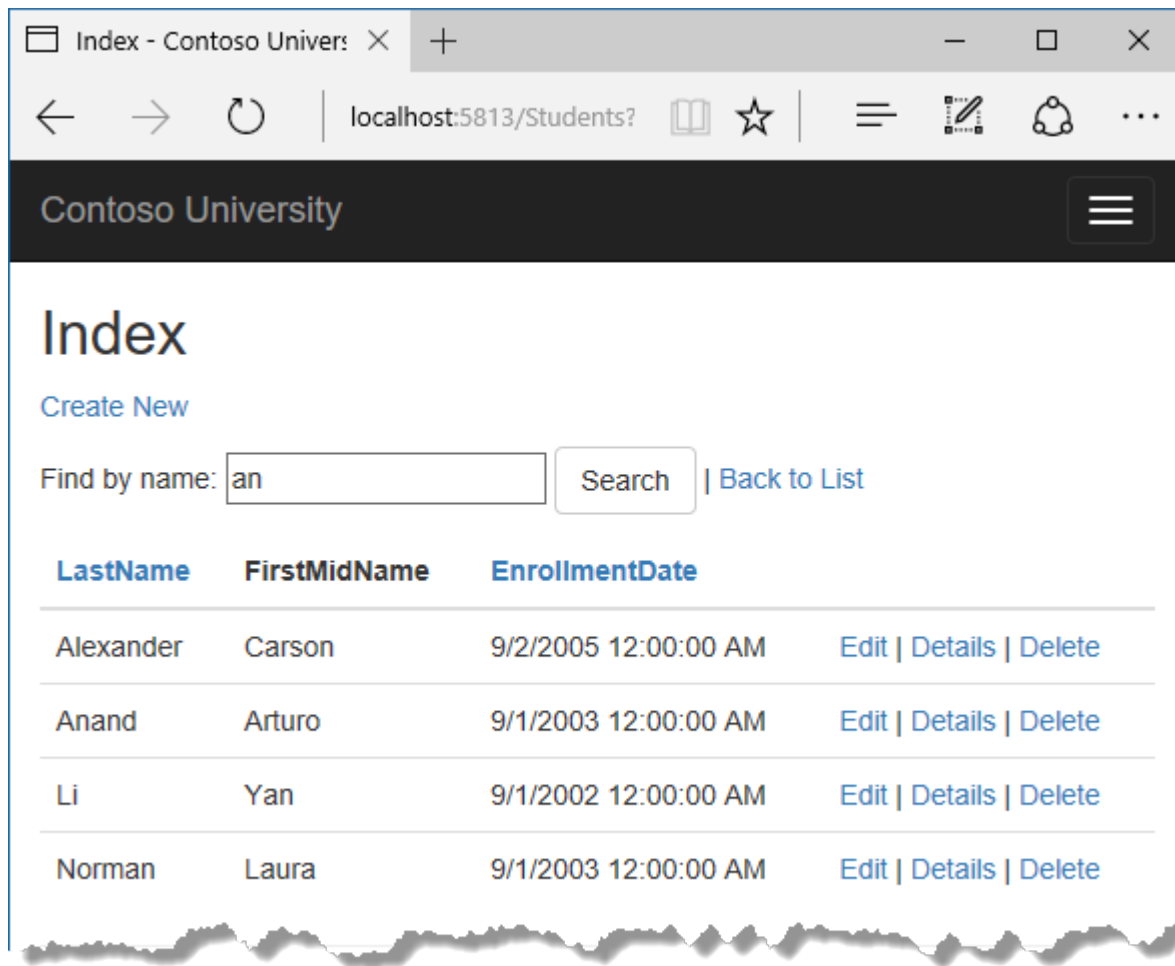
HTML	 Copy
<pre><p> <a asp-action="Create">Create New </p></pre>	

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

<table class="table">
```

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

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



Notice that the URL contains the search string.

HTML

Copy

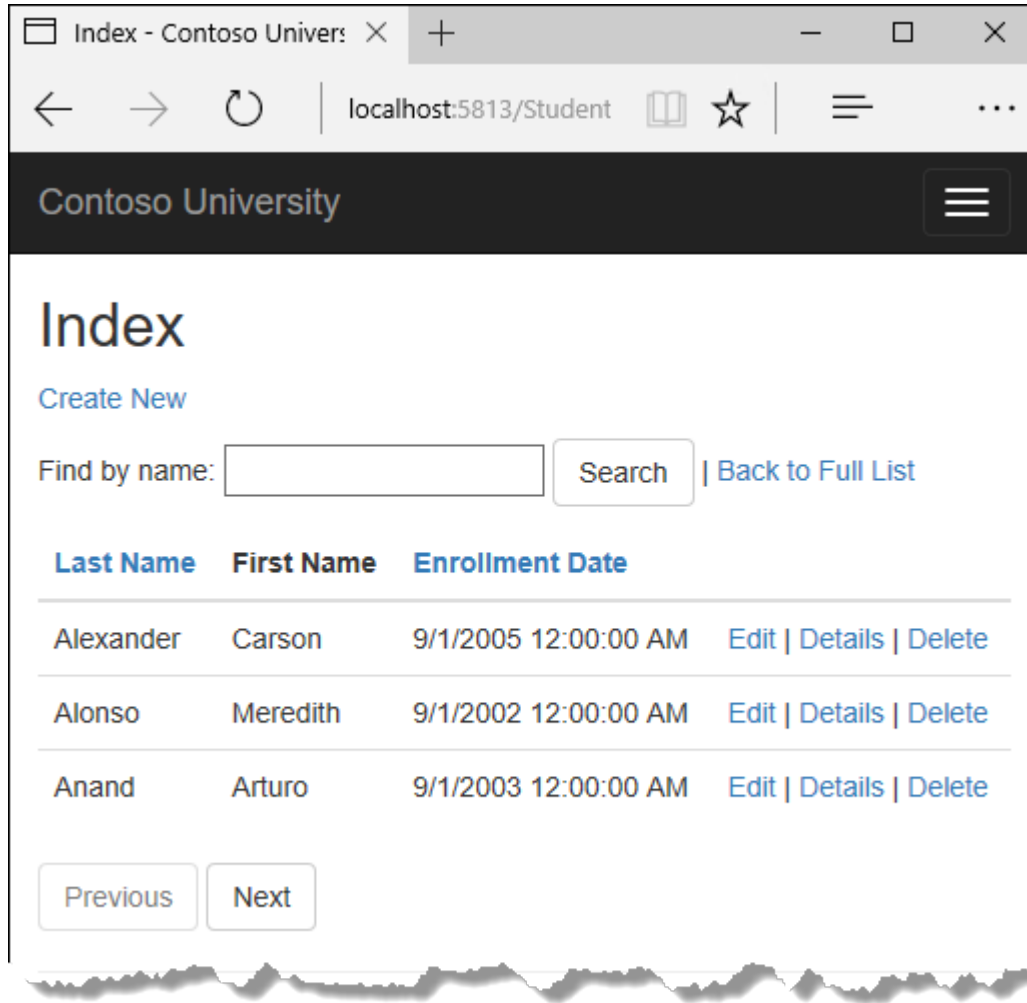
```
http://localhost:5813/Students?SearchString=an
```

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

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

Add paging to Students Index

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



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

C#

Copy

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

```
namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage
        {
            get
            {
                return (PageIndex > 1);
            }
        }

        public bool HasNextPage
        {
            get
            {
                return (PageIndex < TotalPages);
            }
        }

        public static async Task<PaginatedList<T>> CreateAsync(IQueryable<T> source, int pageIndex, int
        pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip((pageIndex - 1) * pageSize).Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}
```

```
}  
}
```

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

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

Add paging to Index method

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

C#

 Copy

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



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

```
ViewData["CurrentFilter"] = searchString;
```

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

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

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

C#

 Copy

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

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

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

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

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

C#


 Copy

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

```
searchString = currentFilter;  
}
```

At the end of the `Index` method, the `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is then passed to the view.

C#

 Copy


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

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

Add paging links

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

HTML

 Copy

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

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

```
<h2>Index</h2>
```

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

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

```

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

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["NameSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Last Name</a>
            </th>
            <th>
                First Name
            </th>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Enrollment Date</a>
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
            </tr>
        }
    </tbody>
</table>

```

```

        </td>
        <td>
            <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
            <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
</tbody>
</table>

```

```

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


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

```

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

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

HTML

 Copy

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

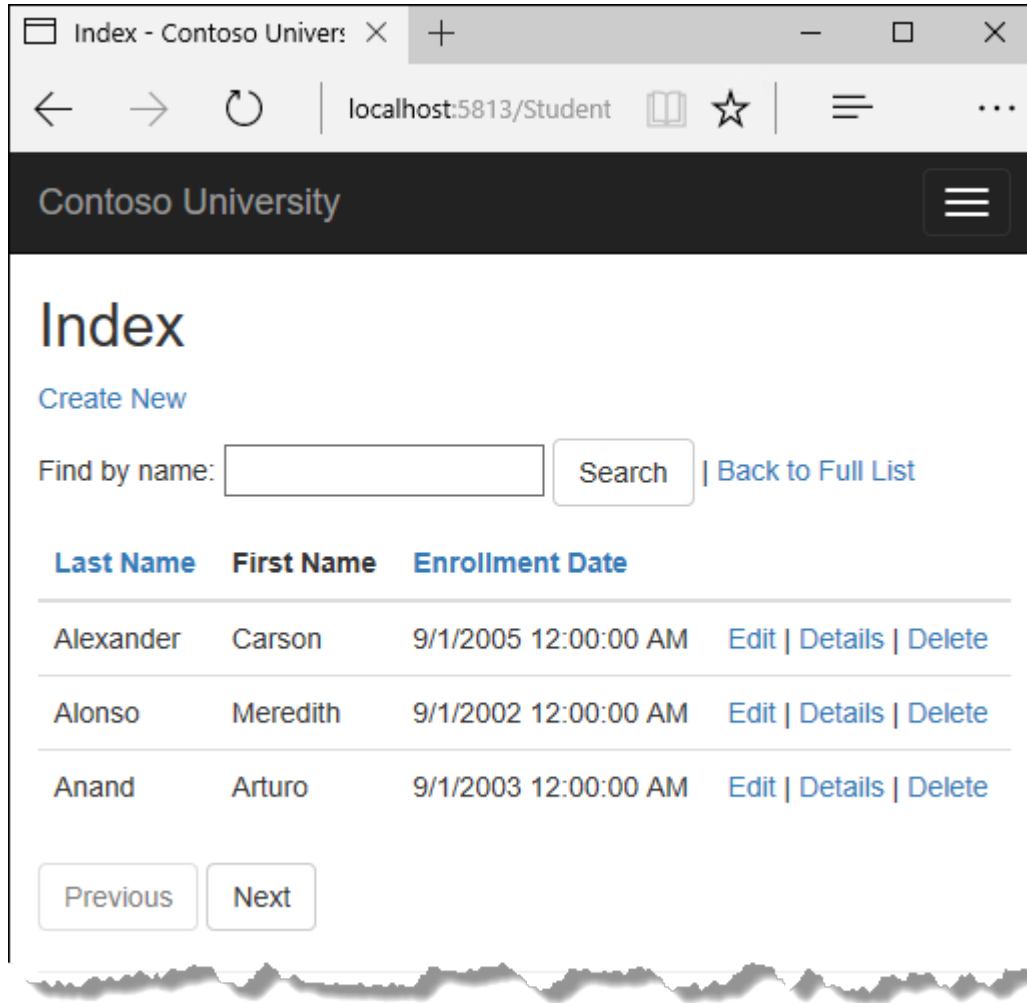
The paging buttons are displayed by tag helpers:

HTML

 Copy

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

Run the app and go to the Students page.



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

Create an About page

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

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

Create the view model

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

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

C#

 Copy

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

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

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

Modify the Home Controller

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

C#

 Copy

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Data;
```



```
using ContosoUniversity.Models.SchoolViewModels;
```

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

C#

 Copy

```
public class HomeController : Controller
{
    private readonly SchoolContext _context;

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

Add an `About` method with the following code:

C#

 Copy


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

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

Create the About View

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

HTML

 Copy

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

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

<h2>Student Body Statistics</h2>

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

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

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

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

Get the code

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

Next steps

In this tutorial, you:

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

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

Next: Handle data model changes

Is this page helpful?

 Yes  No

