

# Add search to an ASP.NET Core MVC app

📅 04/07/2017 ⌚ 7 minutes to read Contributors    

## In this article


[Adding Search by genre](#)

[Adding search by genre to the Index view](#)


By [Rick Anderson](#)

In this section you add search capability to the `Index` action method that lets you search movies by *genre* or *name*.

Update the `Index` method with the following code:

C#	 Copy
<pre>public async Task&lt;IActionResult&gt; Index(string searchString) {     var movies = from m in _context.Movie                  select m;      if (!String.IsNullOrEmpty(searchString))     {         movies = movies.Where(s =&gt; s.Title.Contains(searchString));     }      return View(await movies.ToListAsync()); }</pre>	

The first line of the `Index` action method creates a [LINQ](#) query to select the movies:

C#	 Copy
<pre>var movies = from m in _context.Movie              select m;</pre>	

The query is *only* defined at this point, it has **not** been run against the database.

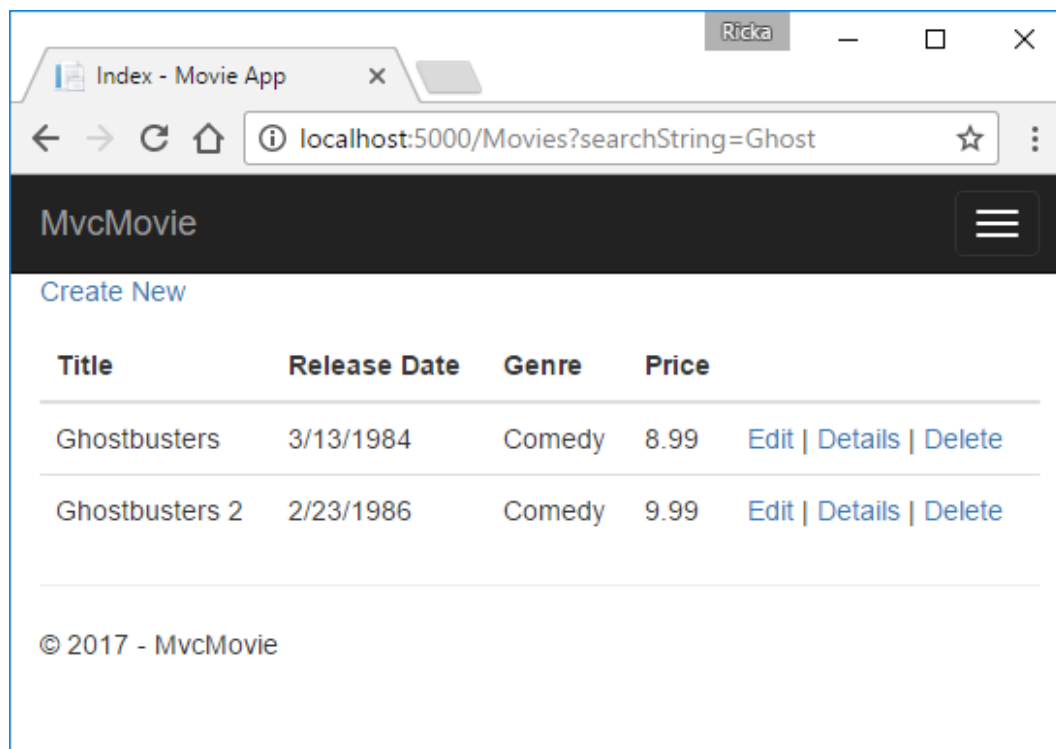
If the `searchString` parameter contains a string, the movies query is modified to filter on the value of the search string:

```
C# Copy  
  
if (!String.IsNullOrEmpty(searchString))  
{  
    movies = movies.Where(s => s.Title.Contains(searchString));  
}
```

The `s => s.Title.Contains()` code above is a [Lambda Expression](#). Lambdas are used in method-based [LINQ](#) queries as arguments to standard query operator methods such as the [Where](#) method or `Contains` (used in the code above). LINQ queries are not executed when they're defined or when they're modified by calling a method such as `Where`, `Contains` or `OrderBy`. Rather, query execution is deferred. That means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToListAsync` method is called. For more information about deferred query execution, see [Query Execution](#).

Note: The [Contains](#) method is run on the database, not in the c# code shown above. The case sensitivity on the query depends on the database and the collation. On SQL Server, [Contains](#) maps to [SQL LIKE](#), which is case insensitive. In SQLite, with the default collation, it's case sensitive.

Navigate to `/Movies/Index`. Append a query string such as `?searchString=Ghost` to the URL. The filtered movies are displayed.



The screenshot shows a web browser window with the title 'Index - Movie App'. The address bar displays 'localhost:5000/Movies?searchString=Ghost'. The page content includes a header 'MvcMovie' with a hamburger menu icon. Below the header is a 'Create New' link. A table lists movies with columns: Title, Release Date, Genre, Price, and actions (Edit, Details, Delete). The table contains two rows: 'Ghostbusters' (3/13/1984, Comedy, 8.99) and 'Ghostbusters 2' (2/23/1986, Comedy, 9.99). At the bottom, there is a copyright notice '© 2017 - MvcMovie'.

Title	Release Date	Genre	Price	
Ghostbusters	3/13/1984	Comedy	8.99	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Ghostbusters 2	2/23/1986	Comedy	9.99	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

© 2017 - MvcMovie


If you change the signature of the `Index` method to have a parameter named `id`, the `id` parameter will match the optional `{id}` placeholder for the default routes set in *Startup.cs*.

C# 

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

Note: SQLite is case sensitive, so you'll need to search for "Ghost" and not "ghost".

The previous `Index` method:


C# 

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                  select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

The updated `Index` method with `id` parameter:

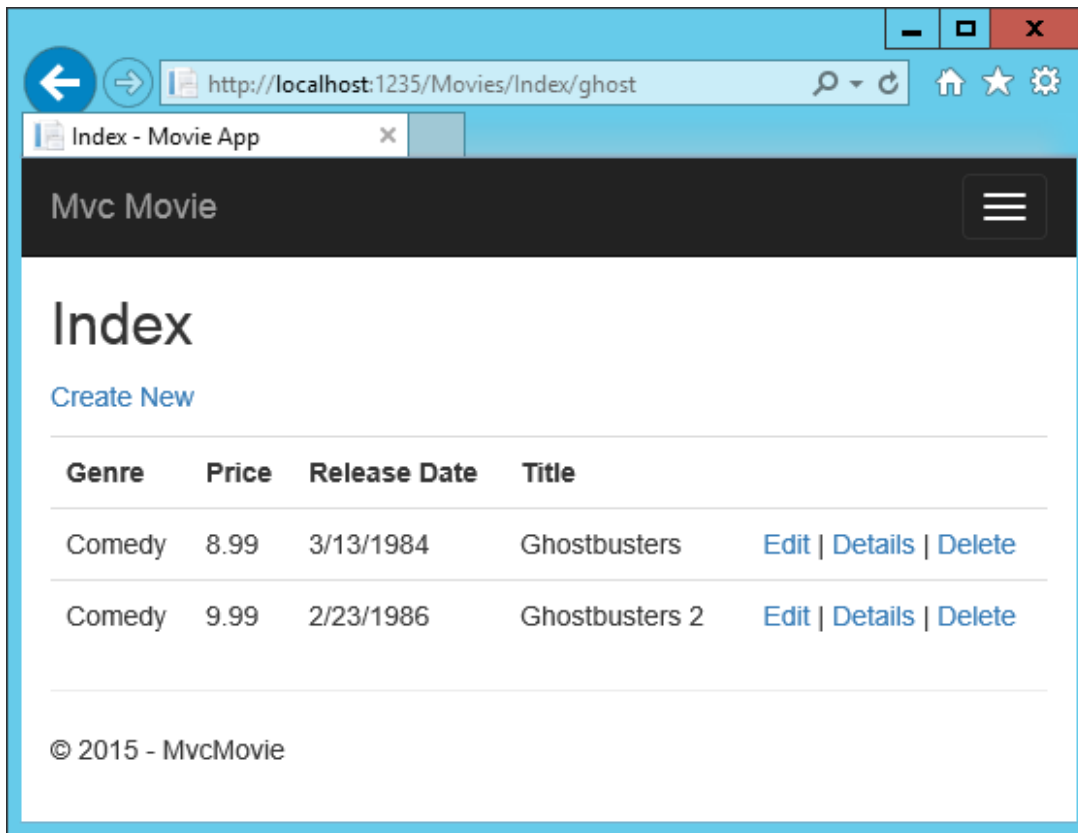
C# 

```
public async Task<IActionResult> Index(string id)
{
    var movies = from m in _context.Movie
                  select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s => s.Title.Contains(id));
    }

    return View(await movies.ToListAsync());
}
```

You can now pass the search title as route data (a URL segment) instead of as a query string value.



However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI elements to help them filter movies. If you changed the signature of the `Index` method to test how to pass the route-bound `ID` parameter, change it back so that it takes a parameter named `searchString`:

```
C# Copy

public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                  select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

Open the `Views/Movies/Index.cshtml` file, and add the `<form>` markup highlighted below:

```
HTML Copy
```

```

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

<h2>Index</h2>

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

<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>

```

The HTML `<form>` tag uses the [Form Tag Helper](#), so when you submit the form, the filter string is posted to the `Index` action of the movies controller. Save your changes and then test the filter.

Index - Movie App × +

localhost:1899/Movies

MvcMovie Window Snip

Create New

Title:

Genre	Price	Release Date	Title	
Romantic Comedy	7.99	1/11/1989	When Harry Met Sally	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Comedy	8.99	3/13/1984	Ghostbusters	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Comedy	9.99	2/23/1986	Ghostbusters 2	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Western	3.99	4/15/1959	Rio Bravo	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

© 2016 - MvcMovie

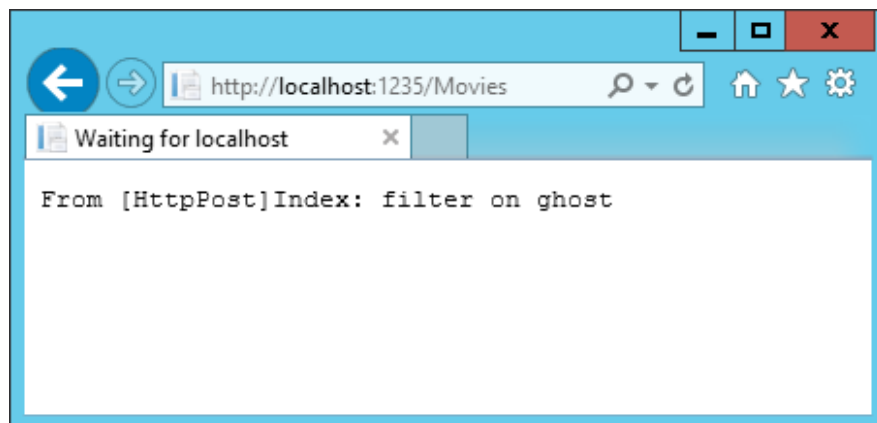
There's no `[HttpPost]` overload of the `Index` method as you might expect. You don't need it, because the method isn't changing the state of the app, just filtering data.

You could add the following `[HttpPost] Index` method.

```
C# Copy  
  
[HttpPost]  
public string Index(string searchString, bool notUsed)  
{  
    return "From [HttpPost]Index: filter on " + searchString;  
}
```

The `notUsed` parameter is used to create an overload for the `Index` method. We'll talk about that later in the tutorial.

If you add this method, the action invoker would match the `[HttpPost] Index` method, and the `[HttpPost] Index` method would run as shown in the image below.



However, even if you add this `[HttpPost]` version of the `Index` method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (`localhost:xxxxx/Movies/Index`) -- there's no search information in the URL. The search string information is sent to the server as a [form field value](#). You can verify that with the browser Developer tools or the excellent [Fiddler tool](#). The image below shows the Chrome browser Developer tools:

Index - Movie App

localhost:5000/Movies

MvcMovie

Create New

Title:

Filter

Release Date

Title

Ghostbusters 3/13/198

Ghostbusters 2/23/1982

© 2017 - MvcMovie

Elements

Console

Sources

Network

Timeline

Profiles

View: [Icons]

Preserve log

Disable cache

Offline

No throttling

Filter

Regex

Hide data URLs

All

XHR

JS

CSS

Img

Media

Font

Doc

WS

Manifest

Other

Name

Movies

jquery.js /lib/jquery/dist

bootstrap.js /lib/bootstrap/dist/js

site.js?v=EWaMeWsJBYWmL2g... /js

Headers

Preview

Response

Cookies

Timing

General

Request URL: http://localhost:5000/Movies

Request Method: POST

Status Code: 200 OK

Remote Address: [::1]:5000

Referrer Policy: no-referrer-when-downgrade

Response Headers

view source

Cache-Control: no-cache

Content-Type: text/html; charset=utf-8

Date: Mon, 10 Apr 2017 00:10:32 GMT

Pragma: no-cache

Server: Kestrel

Transfer-Encoding: chunked

Request Headers

view source

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8

Accept-Encoding: gzip, deflate, br

Accept-Language: en-US,en;q=0.8

Cache-Control: max-age=0

Connection: keep-alive

Content-Length: 201

Content-Type: application/x-www-form-urlencoded

Cookie: .AspNetCore.Antiforgery.txPTUN878m8=CfDJ8B98MxUFL5pAq2aeCj59HP3vvTrLPK1krIeXsJFchnazwWjLRXzWQjTw-tsEJDQr8bQg-xCdy7DbpfcQ-Hi2HAX0in1R838CvFU6oyWz7VIKmiKjUuXI371\_S-YZR0p8dNaP0mTxZX9JRMj\_XzIig; .AspNetCore.Antiforgery.Mkg1\_D\_R5qY=CfDJ8B98MxUFL5pAq2aeCj59HP2tNs0B01ewBFztibCoKKfe2wXo9rL6Z-9YP41jarbKyJ\_I5aQvz9BMWGFpPwwbH71Jw4I8qgC-CkWlyxAsFwKQyP0MYsYab98gk-z\_M4jH1\_Hw90DBZJuBVS80fzF4tm8

Host: localhost:5000

Origin: http://localhost:5000

Referer: http://localhost:5000/Movies

Upgrade-Insecure-Requests: 1

User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36

Form Data

view source

view URL encoded

SearchString: Ghost

\_\_RequestVerificationToken: CfDJ8B98MxUFL5pAq2aeCj59HP1g2HXMD176MabW7uuk2OAGreBb3y0NuF8TMAjxmJCjRFe-2sF50PV1a72IyFC A9Pao3muZ0f4jtjDND1XEagdJk\_g67wBX12qOKI7DLD980GjMjBB\_-5rvRhJuQCroPRw

4 requests | 3.9 KB transferred | Fi...

You can see the search parameter and [XSRF](#) token in the request body. Note, as mentioned in the previous tutorial, the [Form Tag Helper](#) generates an [XSRF](#) anti-forgery token. We're not modifying data, so we don't need to validate the token in the controller method.

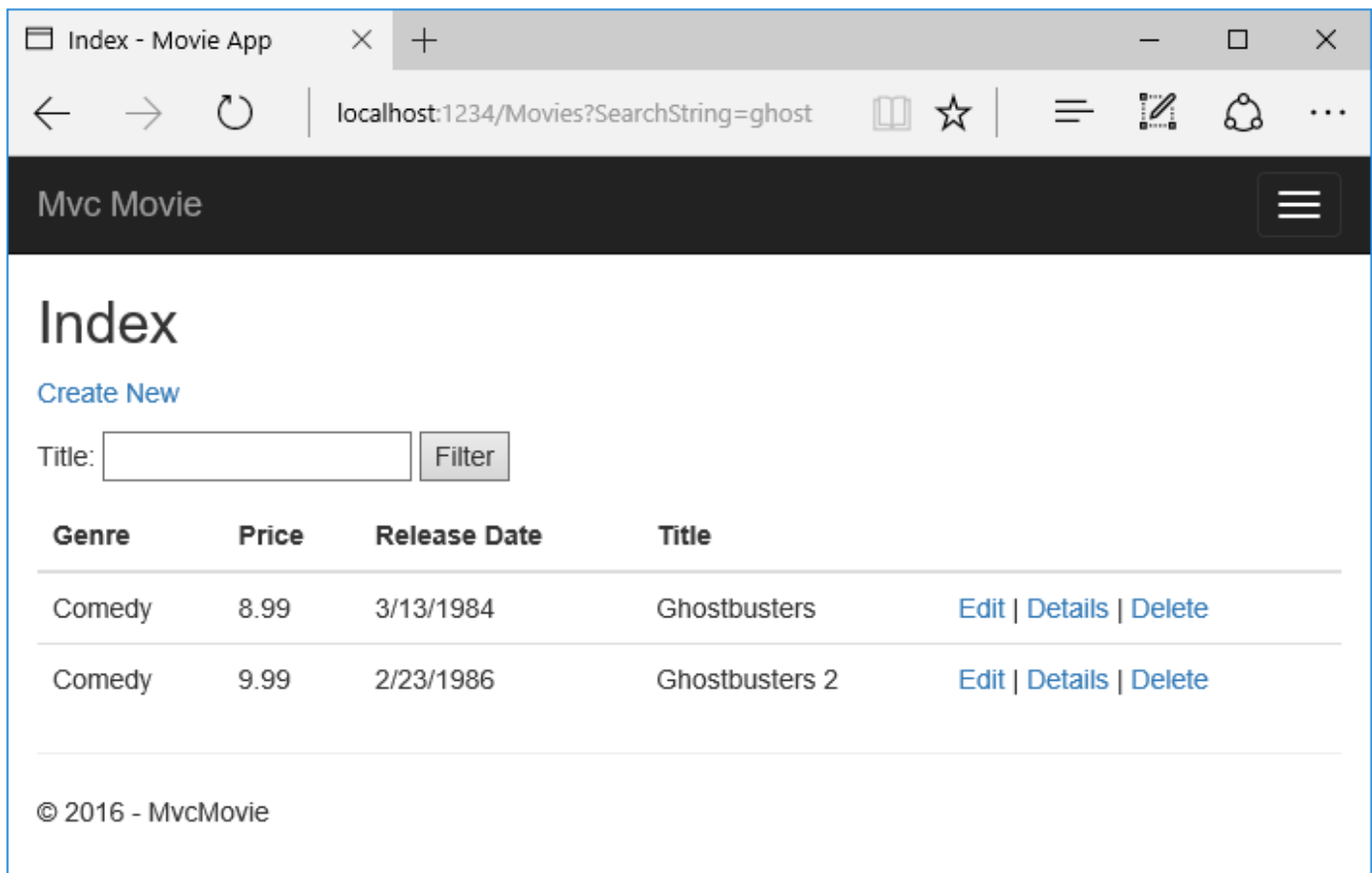
Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. We'll fix this by specifying the request should be `HTTP GET`.

Change the `<form>` tag in the `Views\movie\Index.cshtml` Razor view to specify `method="get"`:

HTML

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

Now when you submit a search, the URL contains the search query string. Searching will also go to the `HttpGet Index` action method, even if you have a `HttpPost Index` method.



The following markup shows the change to the `form` tag:

HTML

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



# Adding Search by genre

Add the following `MovieGenreViewModel` class to the *Models* folder:

C#

 Copy

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace MvcMovie.Models
{
    public class MovieGenreViewModel
    {
        public List<Movie> movies;
        public SelectList genres;
        public string movieGenre { get; set; }
    }
}
```

The movie-genre view model will contain:

- A list of movies.
- A `SelectList` containing the list of genres. This will allow the user to select a genre from the list.
- `movieGenre`, which contains the selected genre.

Replace the `Index` method in `MoviesController.cs` with the following code:

C#

 Copy

```
// Requires using Microsoft.AspNetCore.Mvc.Rendering;
public async Task<IActionResult> Index(string movieGenre, string searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                  select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!String.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }
}
```

```


    }

    var movieGenreVM = new MovieGenreViewModel();
    movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    movieGenreVM.movies = await movies.ToListAsync();


    return View(movieGenreVM);
}

```

The following code is a `LINQ` query that retrieves all the genres from the database.


C#	 Copy
<pre> // Use LINQ to get list of genres. IQueryable&lt;string&gt; genreQuery = from m in _context.Movie                                 orderby m.Genre                                 select m.Genre; </pre>	

The `SelectList` of genres is created by projecting the distinct genres (we don't want our select list to have duplicate genres).

C#	 Copy
<pre> movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync()) </pre>	

## Adding search by genre to the Index view

Update `Index.cshtml` as follows:

HTML	 Copy
<pre> @model MvcMovie.Models.MovieGenreViewModel  @{     ViewData["Title"] = "Index"; }  &lt;h2&gt;Index&lt;/h2&gt;  &lt;p&gt;     &lt;a asp-action="Create"&gt;Create New&lt;/a&gt; &lt;/p&gt;  &lt;form asp-controller="Movies" asp-action="Index" method="get"&gt;     &lt;p&gt;         &lt;select asp-for="movieGenre" asp-items="Model.genres"&gt; </pre>	

```

        <option value="">All</option>
    </select>

    Title: <input type="text" name="SearchString">
    <input type="submit" value="Filter" />
</p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Price)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.movies)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </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>

```

Examine the lambda expression used in the following HTML Helper:

```
@Html.DisplayNameFor(model => model.movies[0].Title)
```

In the preceding code, the `DisplayNameFor` HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. Since the lambda expression is inspected rather than evaluated, you don't receive an access violation when `model`, `model.movies`, or `model.movies[0]` are `null` or empty. When the lambda expression is evaluated (for example, `@Html.DisplayFor(modelItem => item.Title)`), the model's property values are evaluated.

Test the app by searching by genre, by movie title, and by both.

[Previous - Controller methods and views](#)

[Next - Add a field](#)