# Search

By Rick Anderson | last updated May 22, 2015
609 of 655 people found this helpful

## Adding a Search Method and Search View

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

## Updating the Index Form

Start by updating the `Index` action method to the existing `MoviesController` class. Here's the code:

```
public ActionResult Index(string searchString)
{
    var movies = from m in db.Movies
                 select m;

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

    return View(movies);
}
```

The first line of the `Index` method creates the following **LINQ (http://msdn.microsoft.com/en-us/library/bb397926.aspx)** query to select the movies:

```
var movies = from m in db.Movies
                 select m;
```

The query is defined at this point, but hasn't yet 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, using the following code:

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

The `s => s.Title` code above is a **Lambda Expression** (http://msdn.microsoft.com/en-us/library/bb397687.aspx) . Lambdas are used in method-based **LINQ** (http://msdn.microsoft.com/en-us/library/bb397926.aspx) queries as arguments to standard query operator methods such as the **Where** (http://msdn.microsoft.com/en-us/library/system.linq.enumerable.where.aspx) method used in the above code. LINQ queries are not executed when they are defined or when they are modified by calling a method such as `Where` or `OrderBy`. Instead, query execution is deferred, which means that the evaluation of an expression is delayed until its realized value is actually iterated over or the **`ToList`** (http://msdn.microsoft.com/en-us/library/bb342261.aspx) method is called. In the `Search` sample, the query is executed in the *Index.cshtml* view. For more information about deferred query execution, see **Query Execution** (http://msdn.microsoft.com/en-us/library/bb738633.aspx) . **Note**: The **Contains** (http://msdn.microsoft.com/en-us/library/bb155125.aspx) method is run on the database, not the c# code above. On the database, **Contains** (http://msdn.microsoft.com/en-us/library/bb155125.aspx) maps to **SQL LIKE** (http://msdn.microsoft.com/en-us/library/ms179859.aspx) , which is case insensitive.

Now you can update the `Index` view that will display the form to the user.

Run the application and navigate to */Movies/Index*. Append a query string such as `?searchString=ghost` to the URL. The filtered movies are displayed.

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

```
{controller}/{action}/{id}
```

The original `Index` method looks like this::

```
public ActionResult Index(string searchString)
{
    var movies = from m in db.Movies
                 select m;

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

    return View(movies);
}
```

The modified `Index` method would look as follows:

```
public ActionResult Index(string id)
{
    string searchString = id;
    var movies = from m in db.Movies
                 select m;

    if (!String.IsNullOrEmpty(searchString))
```

```
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(movies);
}
```

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 you'll add UI 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 your `Index` method takes a string parameter named `searchString`:

```
public ActionResult Index(string searchString)
{
     var movies = from m in db.Movies
                  select m;

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

    return View(movies);
}
```

Open the *Views\Movies\Index.cshtml* file, and just after `@Html.ActionLink("Create New", "Create")`, add the form markup highlighted below:

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")

     @using (Html.BeginForm()){
         <p> Title: @Html.TextBox("SearchString") <br />
         <input type="submit" value="Filter" /></p>
        }
</p>
```

The `Html.BeginForm (http://msdn.microsoft.com/en-us/library/dd505244(v=vs.108).aspx)` helper creates

an opening <form (http://www.javascript-coder.com/html-form/html-form-tag.phtml) > tag. The Html.BeginForm helper causes the form to post to itself when the user submits the form by clicking the **Filter** button.

Visual Studio 2013 has a nice improvement when displaying and editing View files. When you run the application with a view file open, Visual Studio 2013 invokes the correct controller action method to display the view.

With the Index view open in Visual Studio (as shown in the image above), tap Ctr F5 or F5 to run the application and then try searching for a movie.

There's no HttpPost overload of the Index method. You don't need it, because the method isn't changing the state of the application, just filtering data.

You could add the following HttpPost Index method. In that case, the action invoker would match the HttpPost Index method, and the HttpPost Index method would run as shown in the image below.

```
[HttpPost]
public string Index(FormCollection fc, string searchString)
{
    return "<h3> From [HttpPost]Index: " + searchString + "</h3>";
}
```

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 itself. Right now, the search string information is sent to the server as a form field value. This means you can't capture that search information to bookmark or send to friends in a URL.

The solution is to use an overload of BeginForm (http://msdn.microsoft.com/en-us/library/dd460344(v=vs.108).aspx) that specifies that the POST request should add the search information to the URL and that it should be routed to the HttpGet version of the Index method. Replace the existing parameterless BeginForm method with the following markup:

```
@using (Html.BeginForm("Index","Movies",FormMethod.Get))
```

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

# Adding Search by Genre

If you added the `HttpPost` version of the `Index` method, delete it now.

Next, you'll add a feature to let users search for movies by genre. Replace the `Index` method with the following code:

```
public ActionResult Index(string movieGenre, string searchString)
{
    var GenreLst = new List<string>();

    var GenreQry = from d in db.Movies
                   orderby d.Genre
                   select d.Genre;

    GenreLst.AddRange(GenreQry.Distinct());
    ViewBag.movieGenre = new SelectList(GenreLst);

    var movies = from m in db.Movies
                 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);
    }

    return View(movies);
}
```

This version of the `Index` method takes an additional parameter, namely `movieGenre`. The first few lines of code create a `List` object to hold movie genres from the database.

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

```
var GenreQry = from d in db.Movies
                   orderby d.Genre
                   select d.Genre;
```

The code uses the **AddRange (http://msdn.microsoft.com/en-us/library/z883w3dc.aspx)** method of the generic **List (http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx)** collection to add all the distinct genres to the list. (Without the `Distinct` modifier, duplicate genres would be added — for example, comedy would be added twice in our sample). The code then stores the list of genres in the `ViewBag.movieGenre` object. Storing category data (such a movie genre's) as a **SelectList (http://msdn.microsoft.com/en-us/library/system.web.mvc.selectlist(v=vs.108).aspx)** object in a `ViewBag`, then accessing the category data in a dropdown list box is a typical approach for MVC applications.

The following code shows how to check the `movieGenre` parameter. If it's not empty, the code further constrains the movies query to limit the selected movies to the specified genre.

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

As stated previously, the query is not run on the data base until the movie list is iterated over (which happens in the View, after the `Index` action method returns).

# Adding Markup to the Index View to Support Search by Genre

Add an `Html.DropDownList` helper to the *Views\Movies\Index.cshtml* file, just before the `TextBox` helper. The completed markup is shown below:

```
@model IEnumerable<MvcMovie.Models.Movie>
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
<p>
    @Html.ActionLink("Create New", "Create")
    @using (Html.BeginForm("Index", "Movies", FormMethod.Get))
    {
    <p>
        Genre: @Html.DropDownList("movieGenre", "All")
        Title: @Html.TextBox("SearchString")
        <input type="submit" value="Filter" />
    </p>
    }
</p>
<table class="table">
```

In the following code:

```
@Html.DropDownList("movieGenre", "All")
```

The parameter "movieGenre" provides the key for the `DropDownList` helper to find a `IEnumerable<SelectListItem >` in the `ViewBag`. The `ViewBag` was populated in the action method:

```
public ActionResult Index(string movieGenre, string searchString)
{
    var GenreLst = new List<string>();

    var GenreQry = from d in db.Movies
                   orderby d.Genre
                   select d.Genre;
```

```
    GenreLst.AddRange(GenreQry.Distinct());
    ViewBag.movieGenre = new SelectList(GenreLst);

    var movies = from m in db.Movies
                 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);
    }

    return View(movies);
}
```

The parameter "All" provides the item in the list to be preselected. Had we used the following code:

```
@Html.DropDownList("movieGenre", "Comedy")
```

And we had a movie with a "Comedy" genre in our database, "Comedy" would be preselected in the dropdown list. Because we don't have a movie genre "All", there is no "All" in the `SelectList`, so when we post back without making a slection, the `movieGenre` query string value is empty.

Run the application and browse to */Movies/Index*. Try a search by genre, by movie name, and by both criteria.

In this section you created a search action method and view that let users search by movie title and genre. In the next section, you'll look at how to add a property to the `Movie` model and how to add an initializer that will automatically create a test database.

*This article was originally created on October 17, 2013*

# Author Information

**Rick Anderson** – Rick Anderson works as a programmer writer for Microsoft, focusing on ASP.NET MVC, Windows Azure and Entity Framework. You can follow him on twitter via @RickAndMSFT.

# Comments (79)