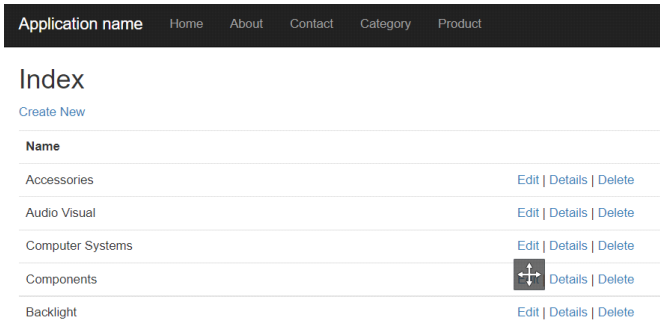
A Simple Query: Sorting Categories Alphabetically

Open CategoriesController.cs and edit the following code in Index method:

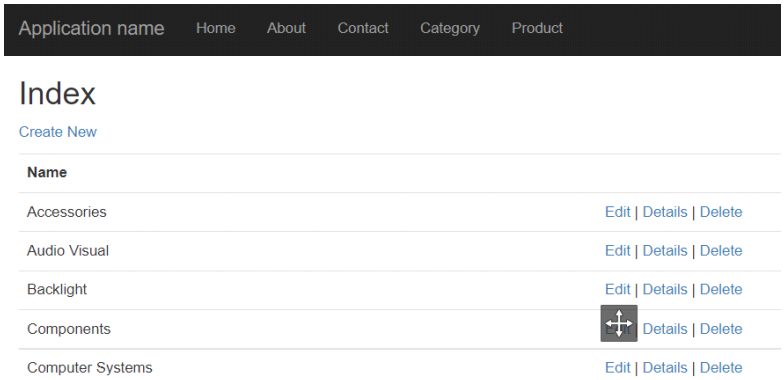
```
// GET: Categories
        public ActionResult Index()
        {
            //return View(db.Categories.ToList());
            return View(db.Categories.OrderBy(c=>c.Name).ToList());
        }
```

This code uses LINQ method syntax to specify which column to order by.
A lambda expression is used to specify the Name column. This code then returns an ordered list of categories to the view for display.
In simple terms, they enable you to create an expression where the value on the left side of the lambda operator (=>) is the input parameter and the value on the right is the expression to be evaluated and returned. Considering the lambda expression that is entered above, it takes a category as an input and returns the Name property. Therefore, in plain English, it says to order by the category's Name property.

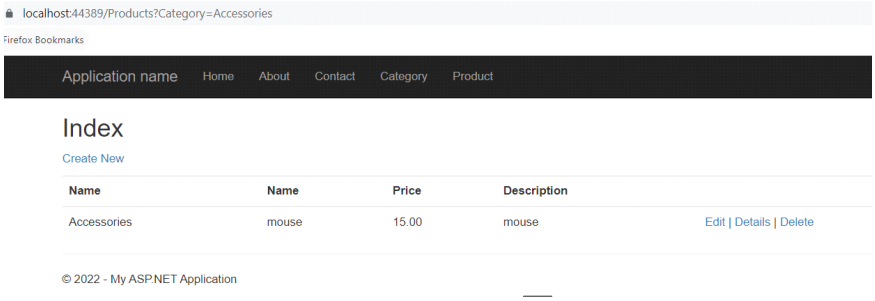


Filtering Products by Category: Searching Related Entities Using Navigational Properties and Include

Now we want to achieve the following:



Using a chosen value from the list of categories to filter the list of products like below:



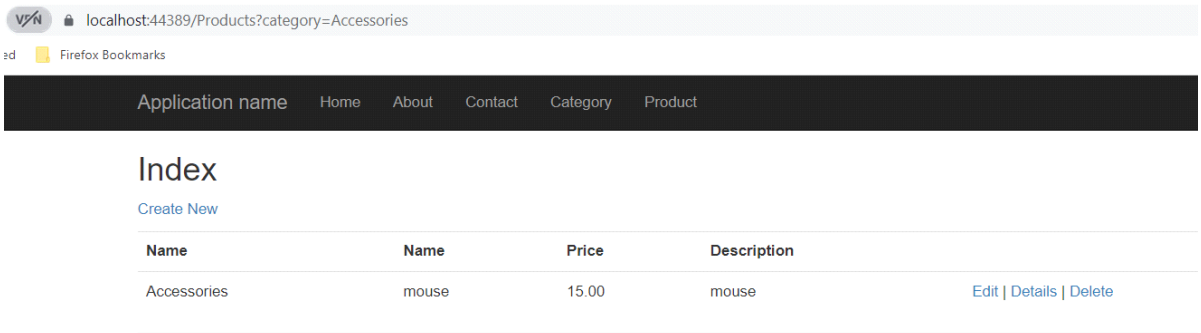
To do this, we'll need to make the following changes to the code:

1. Update the Index method in the ProductsController so that it receives a parameter representing a chosen category and returns a list of products that belong to that category.
2. Transform the list shown in the Category Index Page to a list of hyperlinks that target the ProductsController Index method rather than a list of text items.

First change the ProductsController Index method as follows:

```
// GET: Products
        public ActionResult Index(string category)
        {
            var products = db.Products.Include(p => p.Category);

            if (!String.IsNullOrEmpty(category))
            {
                products = products.Where(p => p.Category.Name == category);
            }
              return View(products.ToList());

        }
```

If you run your code now and add the following address in the url; you will see something like the Figure below:



To change the categories into hyperlinks:

Modify the Views\Categories\Index.cshtml file by
updating the line of code @Html.DisplayFor(modelItem => item.Name) to:
@Html.ActionLink(item.Name, "Index", "Products", new { category = item.Name }, null)

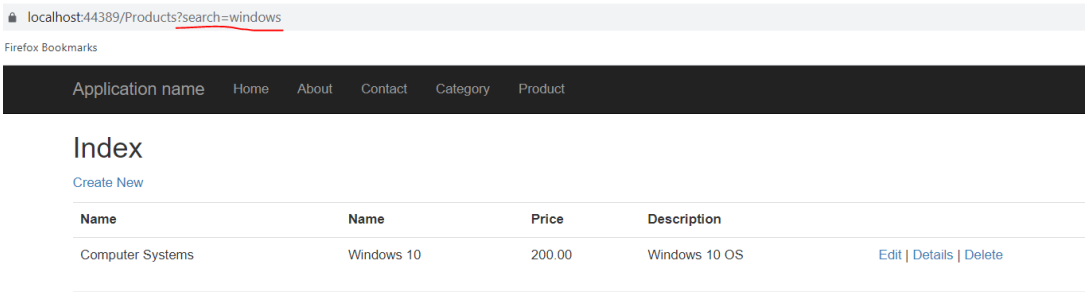
Searching, Advanced Filtering and View Models

Updating the Controller for Product Searching

To add product search, modify the Index method of the Controllers\ProductsController.cs file.

```
  public ActionResult Index(string category, string search)
        {
            var products = db.Products.Include(p => p.Category);
            if (!String.IsNullOrEmpty(category))
            {
                products = products.Where(p => p.Category.Name == category);
            }
//find the products where either the product name field contains search,
//the product description contains search, or the product's category name contains
//search

            if (!String.IsNullOrEmpty(search))
            {
                products = products.Where(p => p.Name.Contains(search) ||
                p.Description.Contains(search) ||
                p.Category.Name.Contains(search));
            }
            return View(products.ToList());
        }
```

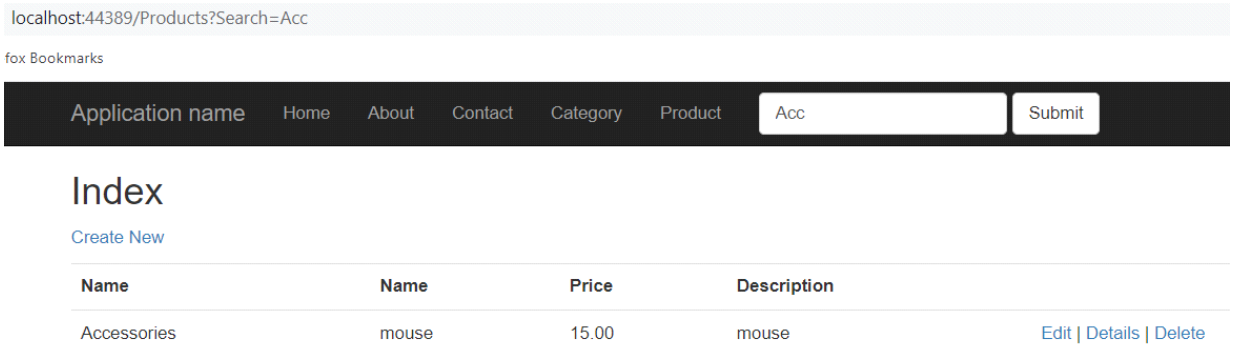
Output: Note the value in Address bar.



Adding a Search Box to the Main Site Navigation Bar

Open file Views\Shared_Layout.cshtml and update the lines in bold within the div tag starting just before the nav:

```
  <div class="navbar-collapse collapse">
          <ul class="nav navbar-nav">
             <li>@Html.ActionLink("Home", "Index", "Home")</li>
             <li>@Html.ActionLink("About", "About", "Home")</li>
             <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
             <li>@Html.ActionLink("Shop by Category", "Index", "Categories")</li>
             <li>@Html.ActionLink("View all our Products", "Index", "Products")</li>
           </ul>
       @using (Html.BeginForm("Index", "Products", FormMethod.Get, new{@class =
"navbar-form navbar-left" }))
       {
              <div class="form-group">
       @Html.TextBox("Search", null, new { @class = "form-control",@placeholder =
"Search Products" })
              </div>
            <button type="submit" class="btn btn-default">Submit</button>
       }

    </div>
```



Updating the ProductsController Index Method to Filter by Category

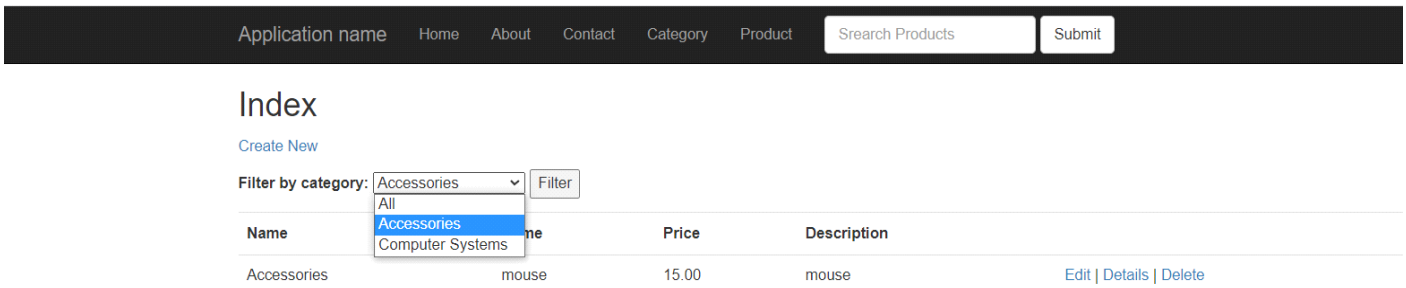
```
public ActionResult Index(string category, string search)
{
      var products = db.Products.Include(p => p.Category);
      if (!String.IsNullOrEmpty(category))
      {
         products = products.Where(p => p.Category.Name == category);
      }

//find the products where either the product name field contains search,the product
//description contains search, or the product's category name contains search
      if (!String.IsNullOrEmpty(search))
      {
         products = products.Where(p => p.Name.Contains(search) ||
         p.Description.Contains(search) ||
         p.Category.Name.Contains(search));
         ViewBag.Search = search; //There is a difference of case in Search/search
      }
    var categories = products.OrderBy (p => p.Category.Name).Select(p
=>p.Category.Name).Distinct();
       ViewBag.Category = new SelectList(categories);

       return View(products.ToList());

 }
```

Adding the Filter to the Products Index Page

Open View\Products\Index.cshtml and update the following code:

```
<p>
    @Html.ActionLink("Create New", "Create");
    @using (Html.BeginForm("Index", "Products", FormMethod.Get))
    {
        <label>Filter by category:</label> @Html.DropDownList("Category", "All")
        <input type="submit" value="Filter" />
        <input type="hidden" name="Search" id="Search" value="@ViewBag.Search" />
    }
</p>
```

The output will be as in the figure below:



However, once the filter is applied, the categories will be removed from the dropdown list. In order to keep seeing all the categories after filtering, you need to change the order in the products controller as below, so that that the products are filtered by category after the categories variable has been populated.

```
 var products = db.Products.Include(p => p.Category);
//find the products where either the product name field contains search, the product
//description contains search, or the product's category name contains search
             if (!String.IsNullOrEmpty(search))
            {
                products = products.Where(p => p.Name.Contains(search) ||
                p.Description.Contains(search) ||
                p.Category.Name.Contains(search));
                ViewBag.Search = search; //There is a difference of case in
Search/search
            }
            var categories = products.OrderBy(p => p.Category.Name).Select(p
=>p.Category.Name).Distinct();

//Move the following lines to after the categories
            if (!String.IsNullOrEmpty(category))
            {
                products = products.Where(p => p.Category.Name == category);
            }

            ViewBag.Category = new SelectList(categories);
            return View(products.ToList());
        }
```

Using a View Model for more Complex Filtering

Creating a View Model:

Create a new folder named ViewModels in the Project, and a new class to it named: ProductIndexViewModel and add the following code:

```
public class ProductIndexViewModel
    {
        public IQueryable<Product> Products { get; set; }
        public string Search { get; set; }
        public IEnumerable<CategoryWithCount> CatsWithCount { get; set; }
        public string Category { get; set; }
```

```
        public IEnumerable<SelectListItem> CatFilterItems
        {

            get
            {
                var allCats = CatsWithCount.Select(cc => new SelectListItem
                {
                    Value = cc.CategoryName,
                    Text = cc.CatNameWithCount
                });
                return allCats;
            }
        }
    }

    public class CategoryWithCount
    {
        public int ProductCount { get; set; }
        public string CategoryName { get; set; }
        public string CatNameWithCount
        {
            get
            {
                return CategoryName + " (" + ProductCount.ToString() + ")";
            }
        }
    }
```

Make sure to add the correct using Statement above to create viewmodel

Update ProductsController.cs

Instead of using ViewBag, now we will use ViewModel.

```
public ActionResult Index(string category, string search)
        {
        //instantiate a new view model
            ProductIndexViewModel viewModel = new ProductIndexViewModel();

            var products = db.Products.Include(p => p.Category);
//find the products where either the product name field contains search,the product
//description contains search, or the product's category name contains search
            if (!String.IsNullOrEmpty(search))
            {
                products = products.Where(p => p.Name.Contains(search) ||
                p.Description.Contains(search) ||
                p.Category.Name.Contains(search));
                // ViewBag.Search = search;
                viewModel.Search = search;
            }
//group search results into categories and count how many items in each category
            viewModel.CatsWithCount = from matchingProducts in products
                                      where
                                      matchingProducts.CategoryID != null
                                      group matchingProducts by
                                      matchingProducts.Category.Name into
                                      catGroup
                                      select new CategoryWithCount()
                                      {
                                          CategoryName = catGroup.Key,
                                          ProductCount = catGroup.Count()
                                      };
```

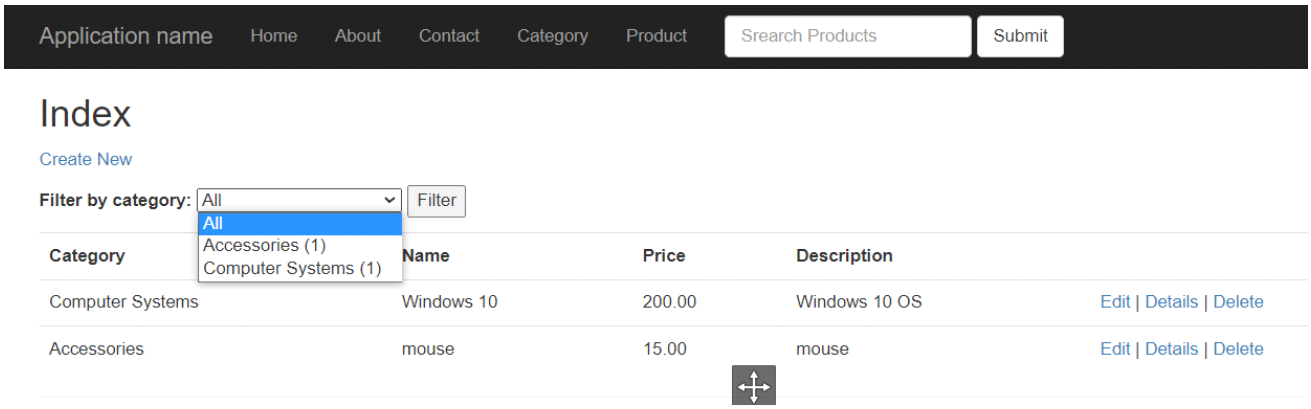
```
//var categories = products.OrderBy(p => p.Category.Name).Select(p =>
p.Category.Name).Distinct();
            if (!String.IsNullOrEmpty(category))
            {
                products = products.Where(p => p.Category.Name == category);
            }
            //ViewBag.Category = new SelectList(categories);
            viewModel.Products = products;
            // return View(products.ToList());
            return View(viewModel);
        }
```

Modifying the View to Display the New Filter Using the View Model

update the \Views\Products\Index.cshtml file

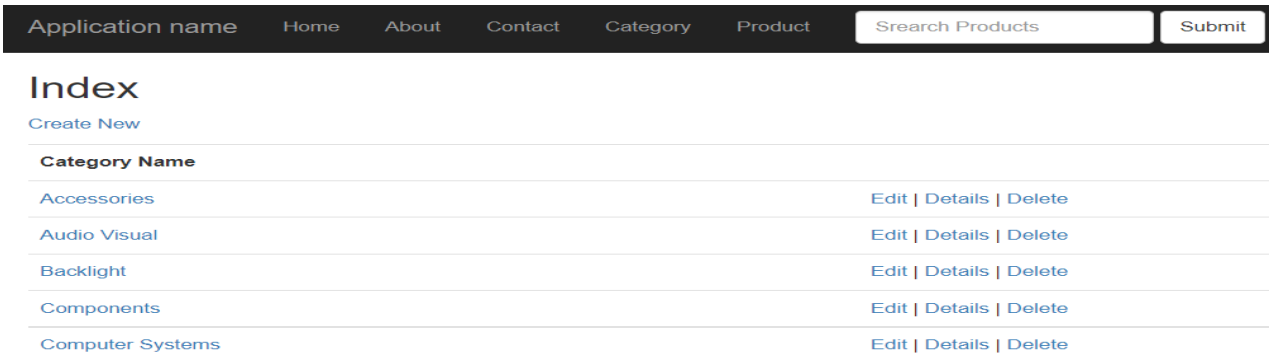
```
@*@model IEnumerable<BasicMVCProject.Models.Product>*@
@model BasicMVCProject.ViewModels.ProductIndexViewModel

@{
    //ViewBag.Title = "Index";
    ViewBag.Title = "Products";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
    @using (Html.BeginForm("Index", "Products", FormMethod.Get))
    {
        <label>Filter by category:</label>
        @*@Html.DropDownList("Category", "All")*@
        @Html.DropDownListFor(vm => vm.Category, Model.CatFilterItems, "All");
        <input type="submit" value="Filter" />
        @*<input type="hidden" name="Search" id="Search" value="@ViewBag.Search" />*@
        <input type="hidden" name="Search" id="Search" value="@Model.Search" />
    }

</p>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Category)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Products.First().Name)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Products.First().Description)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Products.First().Price)
        </th>
        <th></th>
    </tr>

    @foreach (var item in Model.Products)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Category.Name)
            </td>
```

```
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Description)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id = item.ID }) |
                @Html.ActionLink("Details", "Details", new { id = item.ID }) |
                @Html.ActionLink("Delete", "Delete", new { id = item.ID })
            </td>
        </tr>
    }

</table>
```



Now your categories will return with count.

Changing the way, the Category and Product Name properties are displayed using Data Annotations.



In the above Figure, you can see that the Name of the headings has been changed. In our Model, using Annotations, add the line `[Display(Name="Category Name")]` in your Category.cs file code.

```
public class Category
{
    [Key]
    3 references
    public int ID { get; set; }
    [Display(Name="Category Name")]
    8 references
    public string Name { get; set; }
    0 references
    public virtual ICollection<Product> Products { get; set; }
}
```

You may see red underline over it. To remove that, add the right directive as shown in the figure below:

```
public class Category
{
    public int ID { get; set; }
    [Display(Name = "Category Name")]
    pu ic string Name { get; set; }
    pu                                                    t; }
}
```

using System.ComponentModel.DataAnnotations;
System.ComponentModel.DataAnnotations.Display
Generate type 'Display'

CS0246 The type or namespace name 'DisplayAttribute' could not be found (are you missing a using directive or an assembly reference?)

```
...
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
...
```

Preview changes

Similarly, for Product Class:

Application name | Home | About | Contact | Category | Product | Srearch Products | Submit

Index

Create New

Filter by category: All | Filter

Category	Product Name	Price	Description	
Computer Systems	Windows 10	200.00	Windows 10 OS	Edit | Details | Delete
Accessories	mouse	15.00	mouse	Edit | Details | Delete

```
public class Product
{
    [Key]
    3 references
    public int ID { get; set; }
    [Display(Name="Product Name")]
    3 references
    public string Name { get; set; }
    2 references
    public decimal Price { get; set; }
    3 references
    public string Description { get; set; }
    4 references
    public int? CategoryID { get; set; }
    5 references
    public virtual Category Category { get; set; }

}
```