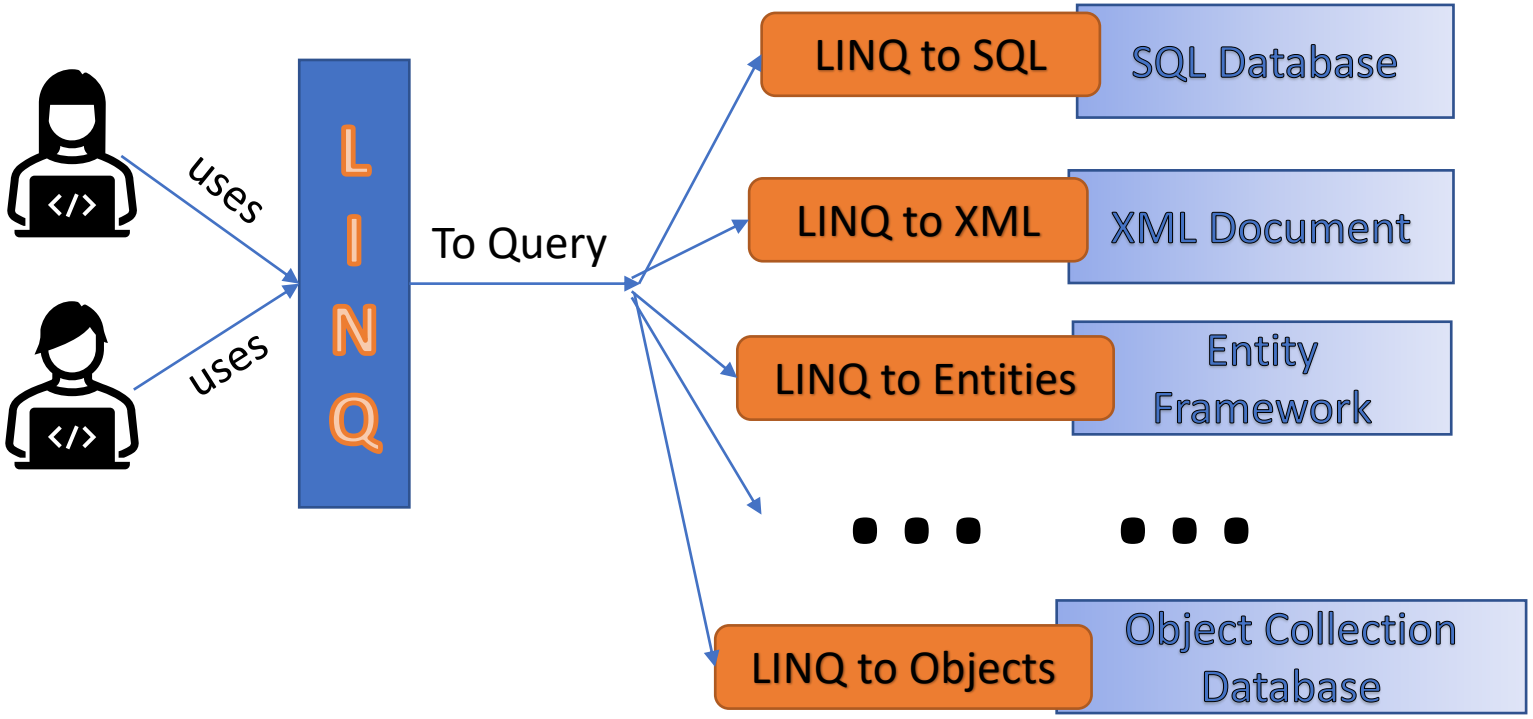
LINQ, Sorting, Searching, Filtering and View Models

What is LINQ?

- Language Integrated Query (LINQ) is a uniform query syntax to save and retrieve data from different sources.



Language-Integrated Query (LINQ)

- LINQ consists of a set of features that allow programmers to query and update data from *various data sources* using the same 'language'.

- LINQ provides a language to communicate with various data sources, independent of the underlying data source technology.

LINQ API

- LINQ is a collection of extension methods for classes that implements IEnumerable and IQueryable interface.
- System.Linq namespace includes the necessary classes and interfaces for LINQ.
- Enumerable and Queryable are two main static classes of LINQ API that contain extension methods.
  - **Enumerable** provides a set of static methods for querying objects that implement IEnumerable<T>
  - **Queryable** provides a set of static methods for querying data structures that implement IQueryable<T>.

Enumerable

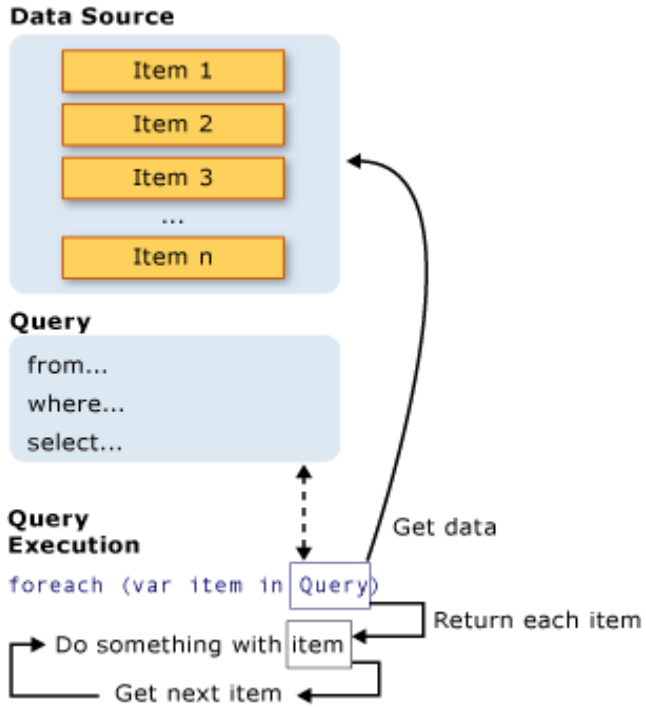
- Enumerable class includes extension methods for the classes that implement IEnumerable<T> interface, this include all the collection types in System.Collections.Generic namespaces such as
  - List<T>,
  - Dictionary<T>,
  - SortedList<T>,
  - Queue<T>,
  - HashSet<T>,
  - LinkedList<T>
  - …etc.

Queryable

- The **Queryable** class includes extension methods for classes that implement IQueryable<T> interface.
- IQueryable<T> is used to provide querying capabilities against a specific data source where the type of the data is known.
  - For example, Entity Framework API implements IQueryable<T> interface to support LINQ queries with underlaying database like SQL Server.
- Also, there are APIs available to access third party data; for example, LINQ to Amazon provides the ability to use LINQ with Amazon web services to search for books and other items by implementing IQueryable interface.

LINQ Queries

- All LINQ query operations consist of three distinct actions:
  - Obtain the data source.
  - Create the query.
  - Execute the query.



The Data Source

- With LINQ to SQL, you first create an object-relational mapping at design time either manually or by using the LINQ to SQL Tools in Visual Studio.
- If the data source is an array, it implicitly supports the generic IEnumerable<T> interface.
- Types that support IEnumerable<T> or a derived interface such as the generic IQueryable<T> are called **queryable types.**
- A queryable type requires no modification or special treatment to serve as a LINQ data source.

The Query

- The query specifies what information to retrieve from the data source or sources.
- Optionally it also specifies how the information should be sorted, grouped and shaped before it is returned.
- A query is stored in a query variable and initialised with a query expression.

Example of Query Syntax in C#

- It contains three clauses:
  - from – specifies the data source.
  - Where – applies the filter.
  - select – specifies the type of the returned elements.

- It is important to note that in LINQ, the query variable itself takes no action and returns no data.
- It only stores the information that is required to produce the results when the query is executed at some later point.

Query Execution

- Deferred Execution
  - The actual execution of the query is deferred until you iterate over the query variable in a foreach statement. This concept is known as deferred execution.

- Forcing Immediate Execution
  - Queries that perform aggregation functions over a range of source elements must first iterate over those elements, e.g. Count, Max, Average, First;
  - these types of queries return a single value and not an IEnumerable collection.

Examples

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```

```
List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();

var numQuery3 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToArray();
```

More Examples

```
List<Student> ls = new List<Student>()
{new Student("teddy", 111, 1.3F),
 new Student("ted", 333, 4.3F),
 new Student("tom", 444, 3.3F),
 new Student("samantha", 555, 3.9F),
 new Student("taylor", 666, 4.0F),
 new Student("joan", 777, 3.1F)
};
```

Example

Task: select all students with an ID smaller than 600

```
List<Student> result = (from s in ls
                        where s.ID < 600
                        select s).ToList();
```

LINQ Example

Task: delete a student with an ID of 444

```
ls = ls.Except(from s in ls
               where s.ID == 444
               select s).ToList();
```

LINQ Example

Task: order all students in a descending order of GPA

```
IEnumerable<Student> ie =
          (from s in ls
           orderby s.GPA descending
           select s);
```

LINQ Example

Task: calculate the average GPA for all students

```
float result = (from s in ls
                select s.GPA).Average();
```

LINQ Example

Task: count all students with an ID greater than 200

```
int result = (from s in ls
              where s.ID > 200
              select s).Count();
```

Examples

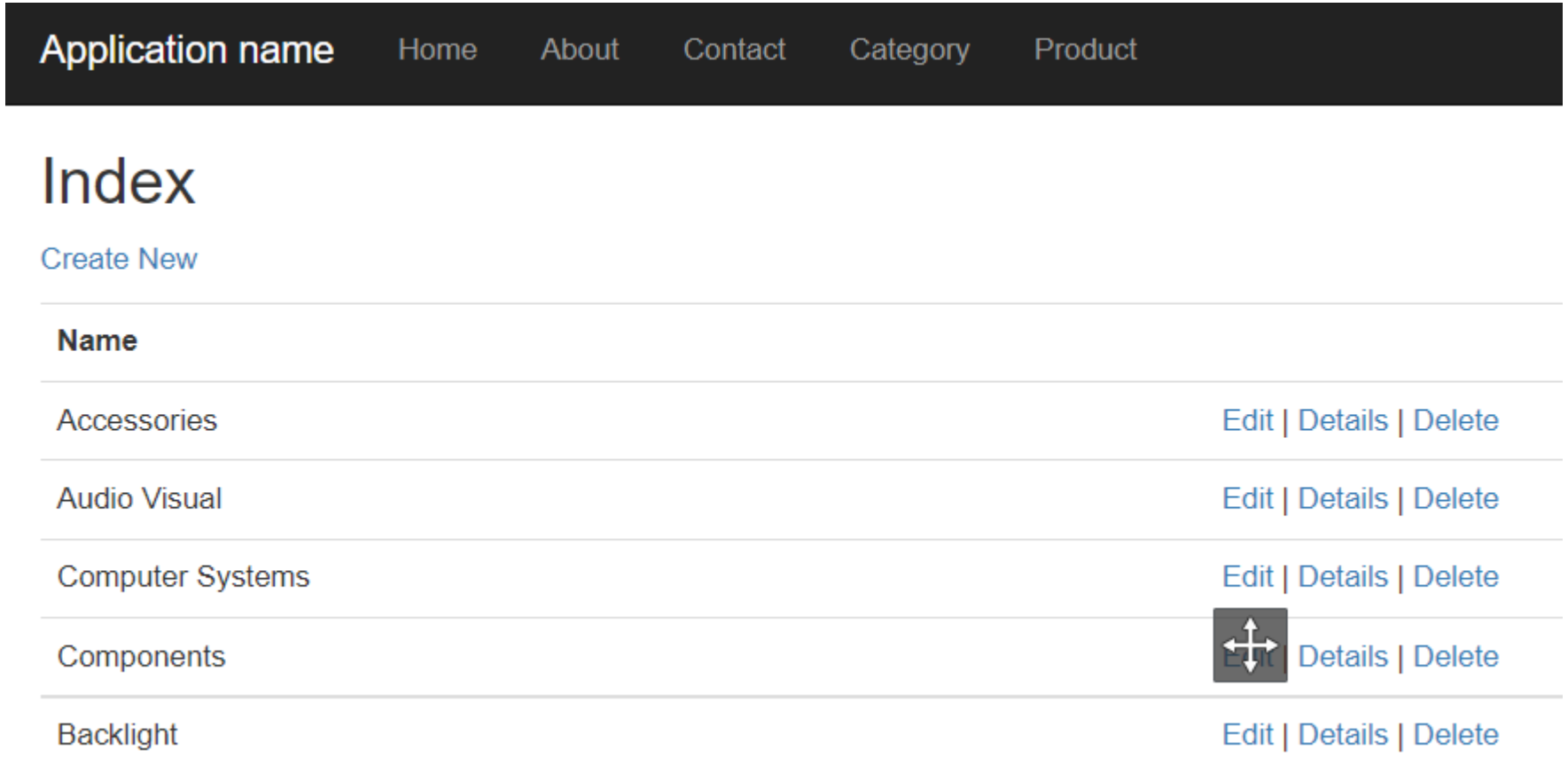
```
Student[] studentArray = {  new Student() { StudentID = 1, StudentName = "John", age = 18 } ,
                            new Student() { StudentID = 2, StudentName = "Steve",  age = 21 } ,
                            new Student() { StudentID = 3, StudentName = "Bill",  age = 25 } ,
                            new Student() { StudentID = 4, StudentName = "Ram" , age = 20 } ,
                            new Student() { StudentID = 5, StudentName = "Ron" , age = 31 } ,
                            new Student() { StudentID = 6, StudentName = "Chris",  age = 17 } ,
                            new Student() { StudentID = 7, StudentName = "Rob",age = 19  }  };
```

```
Student[] teenAgerStudents = studentArray.Where(s => s.age > 12 && s.age < 20).ToArray();
```

```
Student bill = studentArray.Where(s => s.StudentName == "Bill").FirstOrDefault();
```

```
Student student5 = studentArray.Where(s => s.StudentID == 5).FirstOrDefault();
```

A Simple Query: Sorting Categories Alphabetically



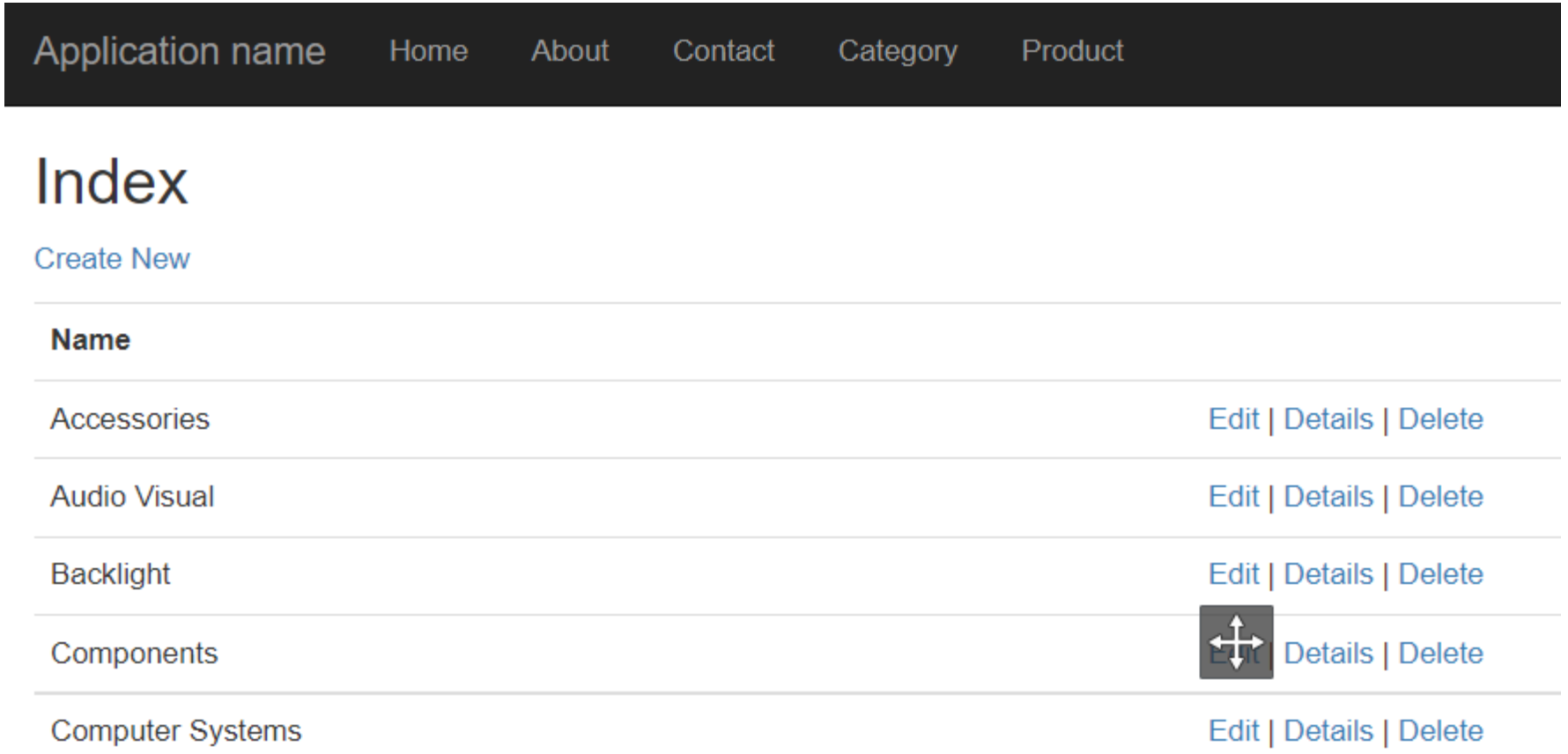
CategoriesController: Index()

```
 public ActionResult Index()
        {
            return View(db.Categories.ToList());
        }
=>

 public class CategoriesController : Controller
    {
        …

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

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



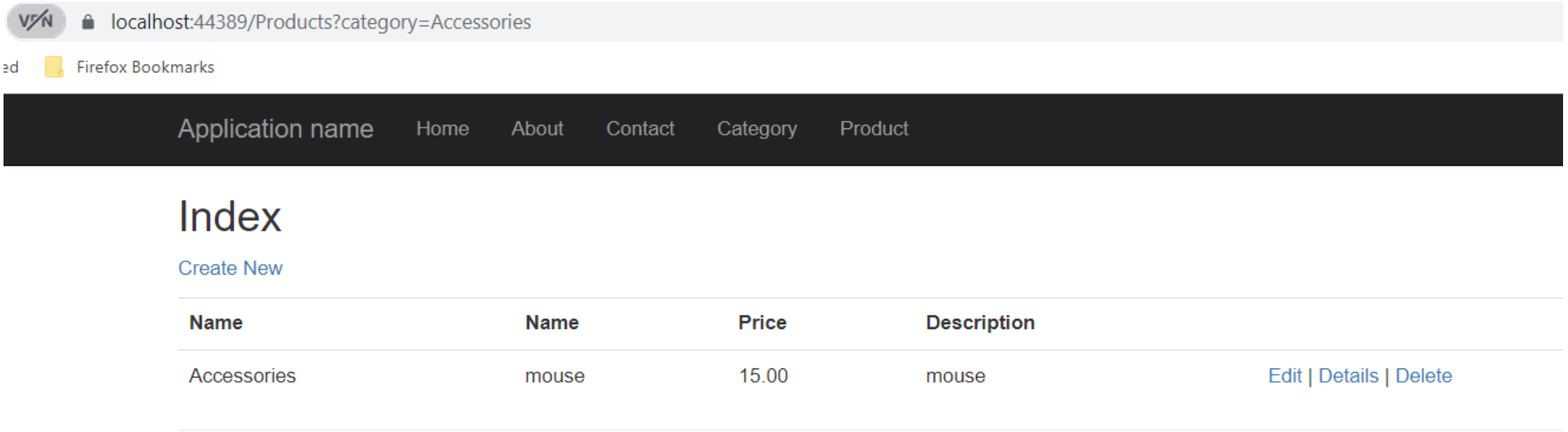
ProductsController: Index()

```
public class ProductsController : Controller
    {
        …
       public ActionResult Index()
        {
            var products = db.Products.Include(p => p.Category);
            return View(products.ToList());
        }
```

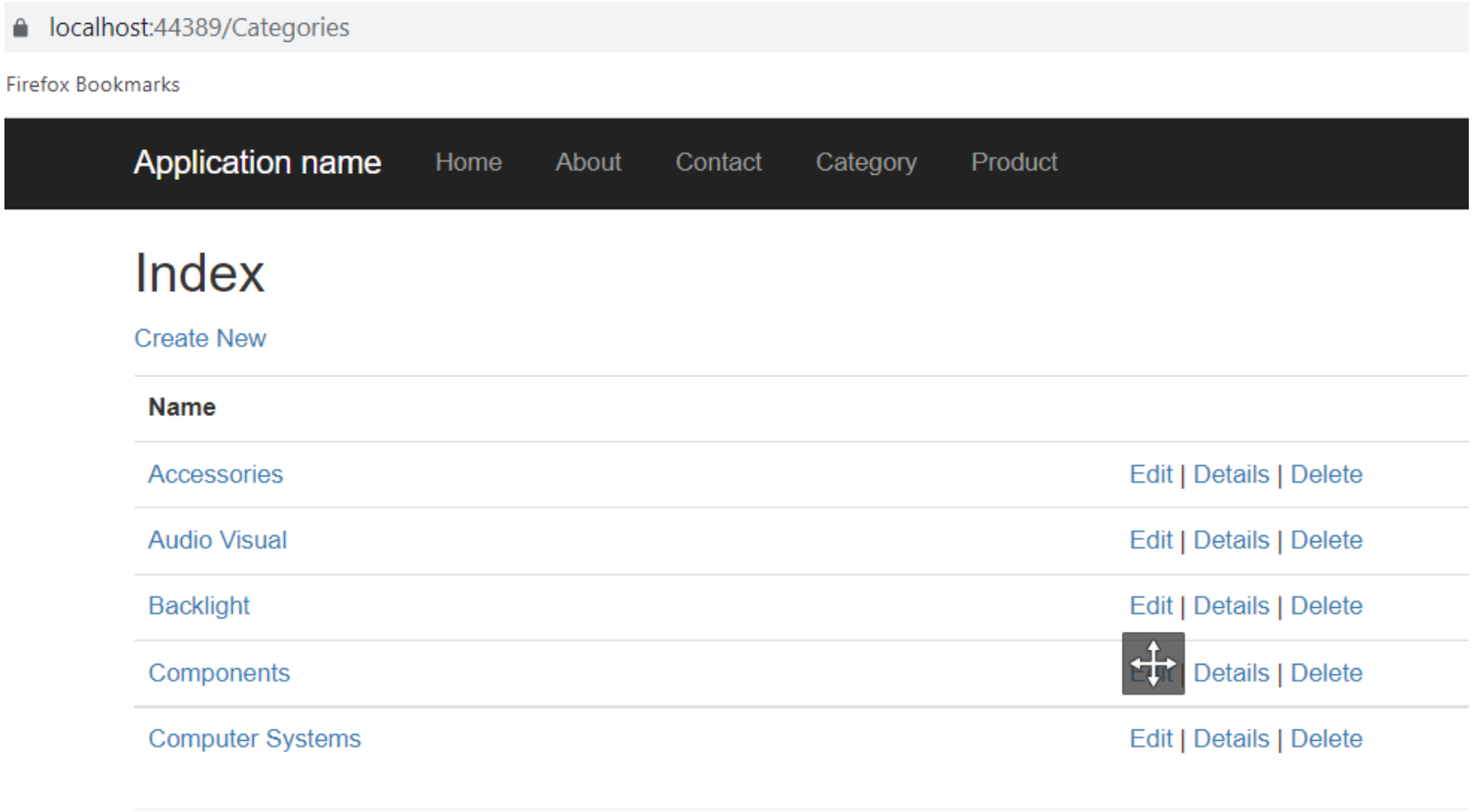
ProductsController : Index(string category)

```
public class ProductsController : Controller
    {
…
        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());
        }
```

URL: ?category=Accessories



Transform the list shown in the Category Index Page to a list of hyperlinks



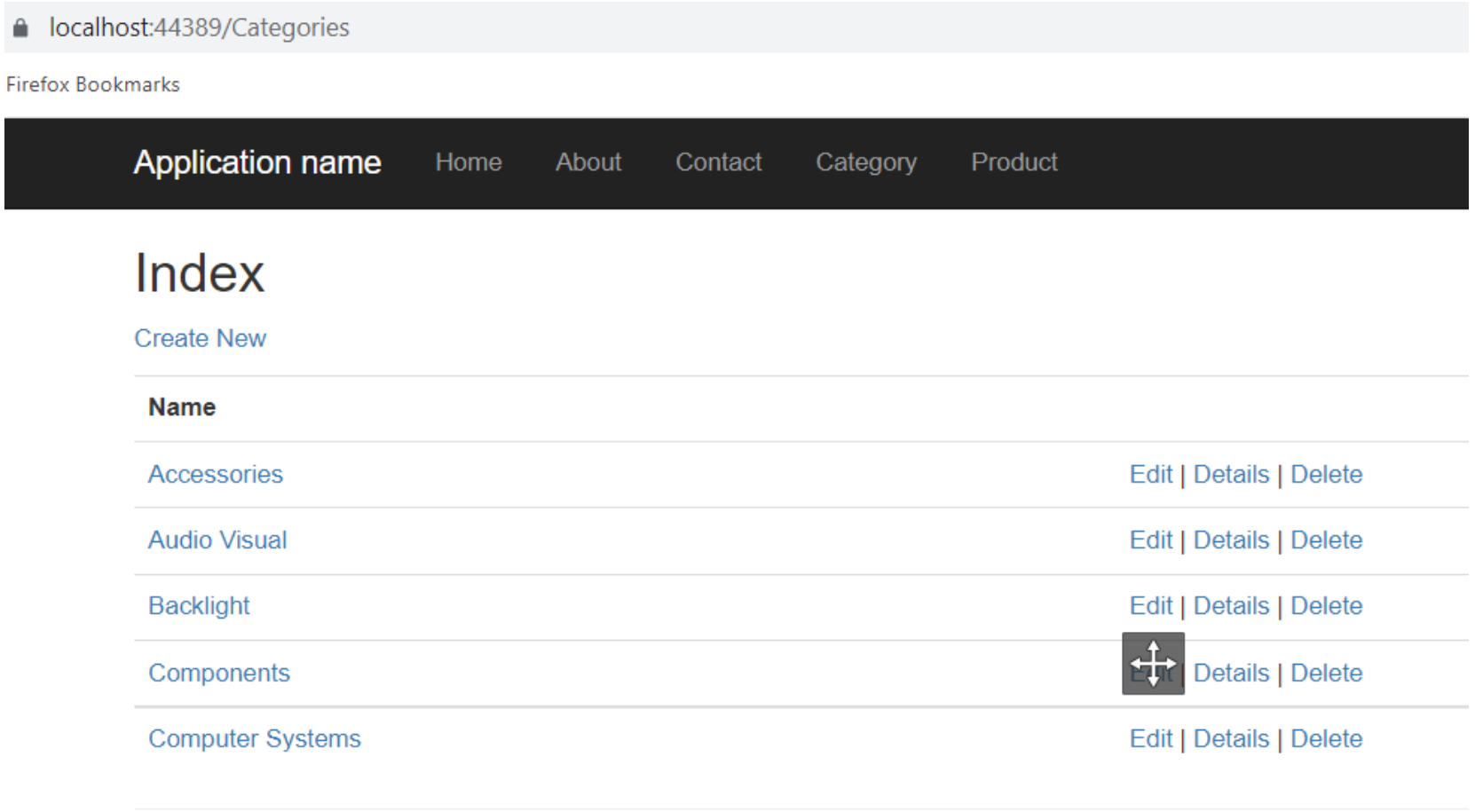
Views\Categories\Index.cshtml

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

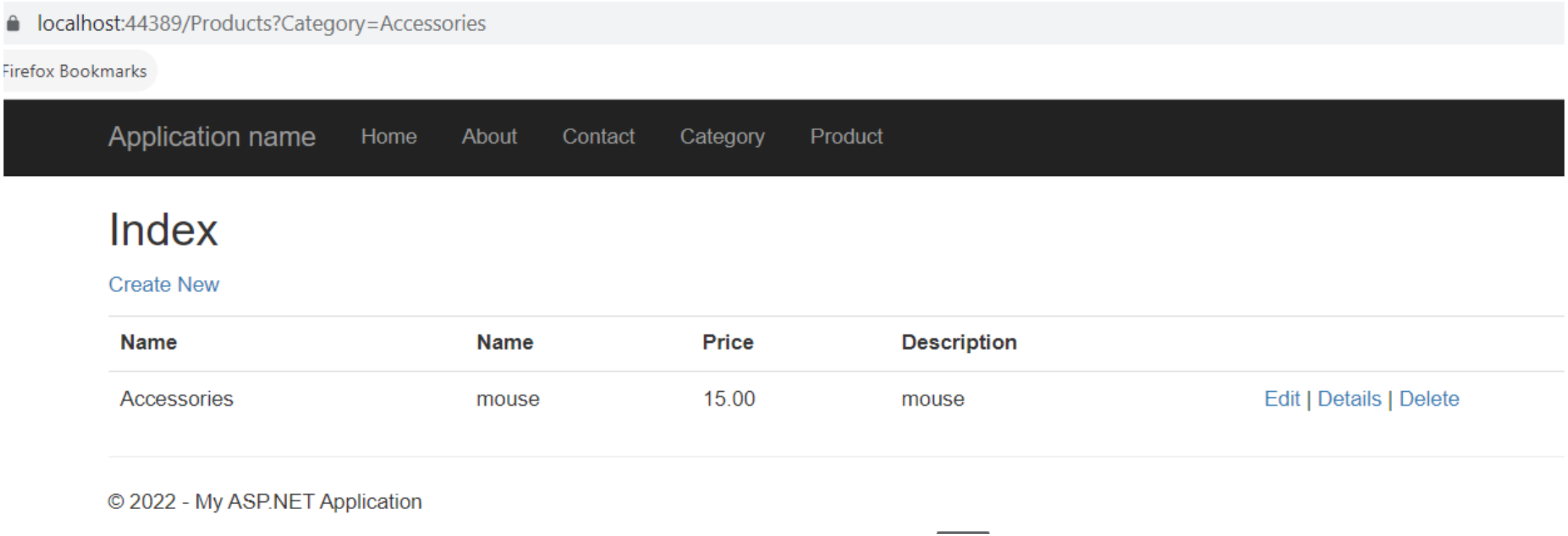
=>

@Html.ActionLink(item.Name, "Index", "Products", new {Category = item.Name}, null)

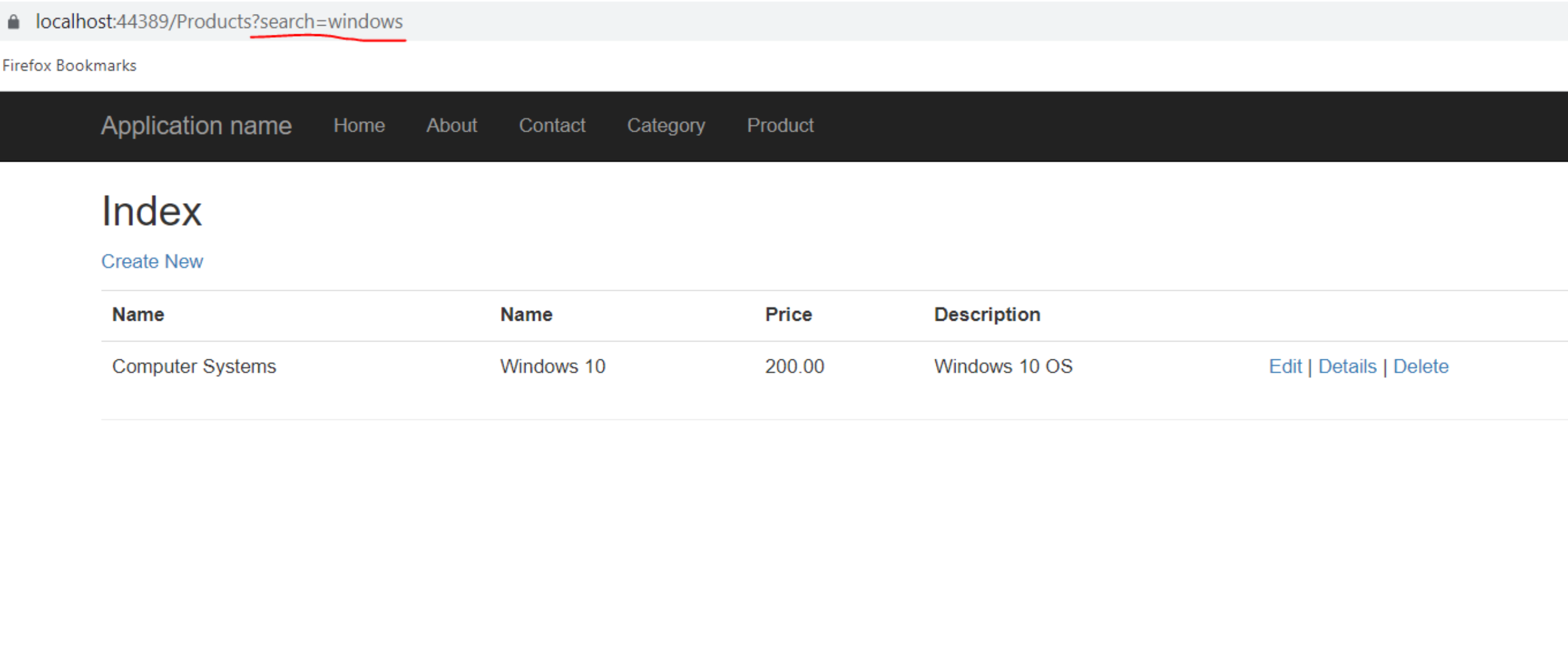
Transform the list shown in the Category Index Page to a list of hyperlinks



After clicking **Accessories** in the Category Index Page



Search for products

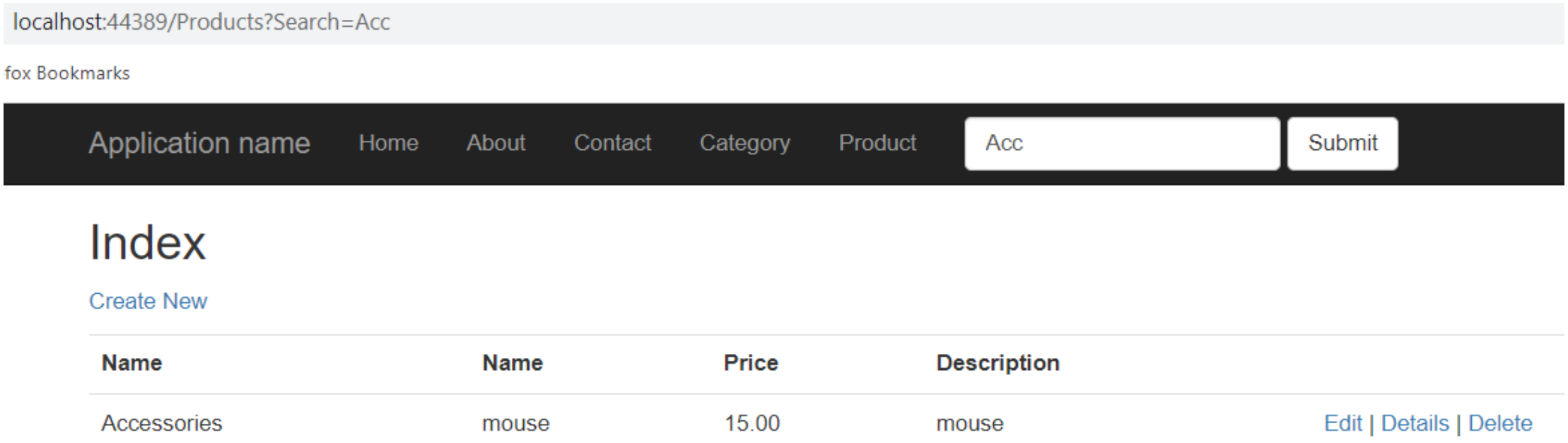


ProductsController

```
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);
            }
            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());
}
```

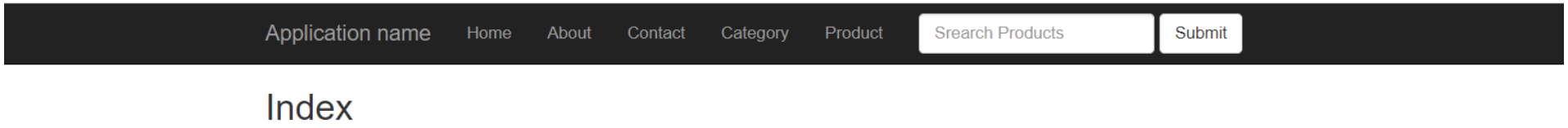
Adding a Search Box to the Main Site Navigation Bar



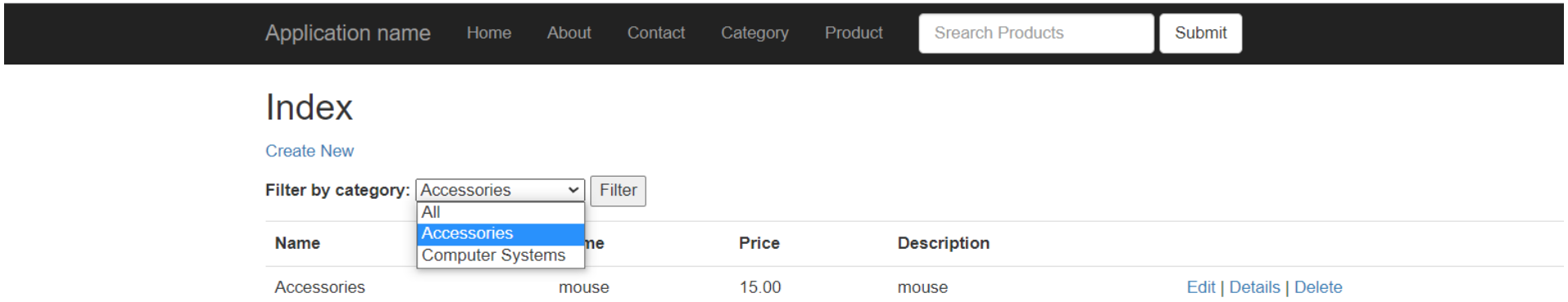
_Layout.cshtml

```
<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("Category", "Index", "Categories")</li>
      <li>@Html.ActionLink("Product", "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 ="Srearch Products"})
        </div>
        <button type="submit" class="btn btn-default">Submit</button>
    }
</div>
```

Navigation bar complete with search box



Filtering the Search Results by Category Using ViewBag



Updating the ProductsController Index Method to Filter by Category

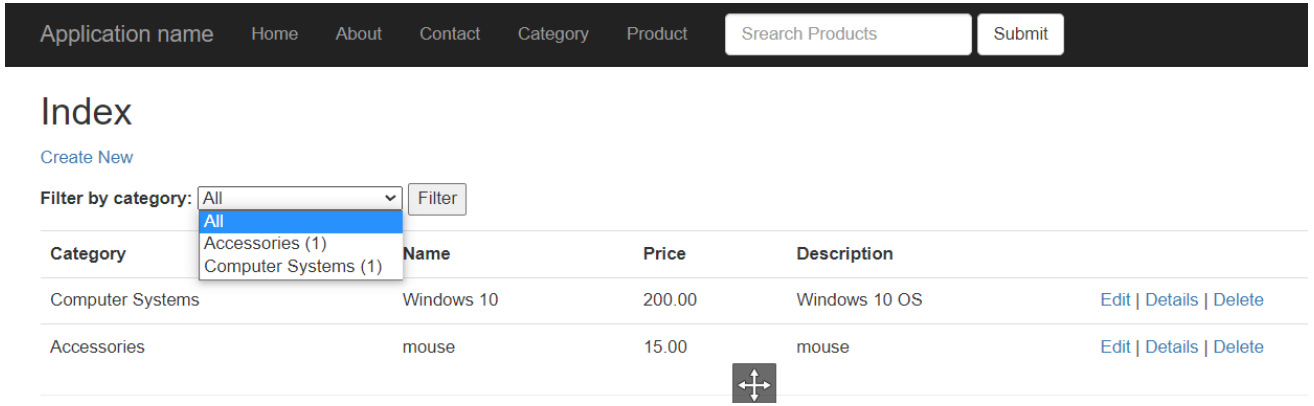
```
public ActionResult Index(string category, string search)
        { …
            if (!String.IsNullOrEmpty(search))
            {
                products = products.Where(p => p.Name.Contains(search) ||
                p.Description.Contains(search) ||
                p.Category.Name.Contains(search));
                ViewBag.Search = search;
            }
            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);
            return View(products.ToList());
        }
```

Adding the Filter to the Products Index Page: View\Products\Index.cshtm

```
@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" />
    }
```

Using a View Model for More Complex Filtering



In ASP.NET MVC, ViewModel is **a class that contains the fields which are represented in the strongly-typed view**. It is used to pass data from controller to strongly-typed view.

Creating a View Model

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

ProductsController

```
public ActionResult Index(string category, string search)
        {
            ProductIndexViewModel viewModel = new ProductIndexViewModel();
            var products = db.Products.Include(p => p.Category);

            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;
            }
```

ProductsController

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

Modifying the View to Display the New Filter Using the View Model: View\Products\Index.cshtm

```
@model BasicMVCProject.ViewModels.ProductIndexViewModel


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

Modifying the View to Display the New Filter Using the View Model: View\Products\Index.cshtm

```
<tr>
        <th>
            @Html.DisplayNameFor(model => model.Category)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Products.First().Name)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Products.First().Price)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Products.First().Description)
        </th>
        <th></th>
    </tr>
```