In the previous chapter, we set up the core infrastructure of the SportsStore application. Now we will use the infrastructure to add key features to the application, and you’ll start to see how the investment in the basic plumbing pays off. We will be able to add important customer-facing features simply and easily. Along the way, you’ll see some additional features that the MVC Framework provides.

Adding Navigation Controls

The SportsStore application will be a lot more usable if we let customers navigate products by category. We will do this in three parts:

- Enhance the List action model in the ProductController class so that it is able to filter the Product objects in the repository.
- Revisit and enhance our URL scheme and revise our rerouting strategy.
- Create the category list that will go into the sidebar of the site, highlighting the current category and linking to others.

Filtering the Product List

We are going to start by enhancing our view model class, ProductsListViewModel. We need to communicate the current category to the view in order to render our sidebar, and this is as good a place to start as any. Listing 8-1 shows the changes we made.

Listing 8-1. Enhancing the ProductsListViewModel Class

```csharp
using System.Collections.Generic;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Models {
    public class ProductsListViewModel {
        public IEnumerable<Product> Products { get; set; }
        public PagingInfo PagingInfo { get; set; }
    }
} 
```
public string CurrentCategory { get; set; }
}

We added a new property called CurrentCategory. The next step is to update the ProductController class so that the List action method will filter Product objects by category and use the new property we added to the view model to indicate which category has been selected. The changes are shown in Listing 8-2.

Listing 8-2. Adding Category Support to the List Action Method

```csharp
public ViewResult List(string category, int page = 1) {
    ProductsListViewModel viewModel = new ProductsListViewModel {
        Products = repository.Products
            .Where(p => category == null || p.Category == category)
            .OrderBy(p => p.ProductID)
            .Skip((page - 1) * PageSize)
            .Take(PageSize),
        PagingInfo = new PagingInfo {
            CurrentPage = page,
            ItemsPerPage = PageSize,
            TotalItems = repository.Products.Count() 
        },
        CurrentCategory = category
    };
    return View(viewModel);
}
```

We've made three changes to this method. First, we added a new parameter called category. This category is used by the second change, which is an enhancement to the LINQ query—if category isn't null, only those Product objects with a matching Category property are selected. The last change is to set the value of the CurrentCategory property we added to the ProductsListViewModel class. However, these changes mean that the value of TotalItems is incorrectly calculated—we’ll fix this in a while.

UNIT TEST: UPDATING EXISTING UNIT TESTS

We have changed the signature of the List action method, which will prevent some of our existing unit test methods from compiling. To address this, pass null as the first parameter to the List method in those unit tests that work with the controller. For example, in the Can_Send_Pagination_View_Model test, the action section of the unit test becomes as follows:

```csharp
ProductsListViewModel result = (ProductsListViewModel)controller.List(null, 2).Model;
```

By using null, we receive all of the Product objects that the controller gets from the repository, which is the same situation we had before we added the new parameter.