Using MVVM in MVC Applications – Part 2

This blog post continues from where the last blog post left off. You are going to learn to search for products. You also learn how to handle all post backs through a single method in your MVC controller. You will add code to check for no rows being returned, and display a message to the user. Finally you break up the single page into multiple partial pages.

Search for Products

You are now going to add a text box to allow the user to fill in a partial product name to search on. You are going to add two new buttons; one to allow the user to search on the product name they fill in, and one to clear the product name text box as shown in Figure 19.

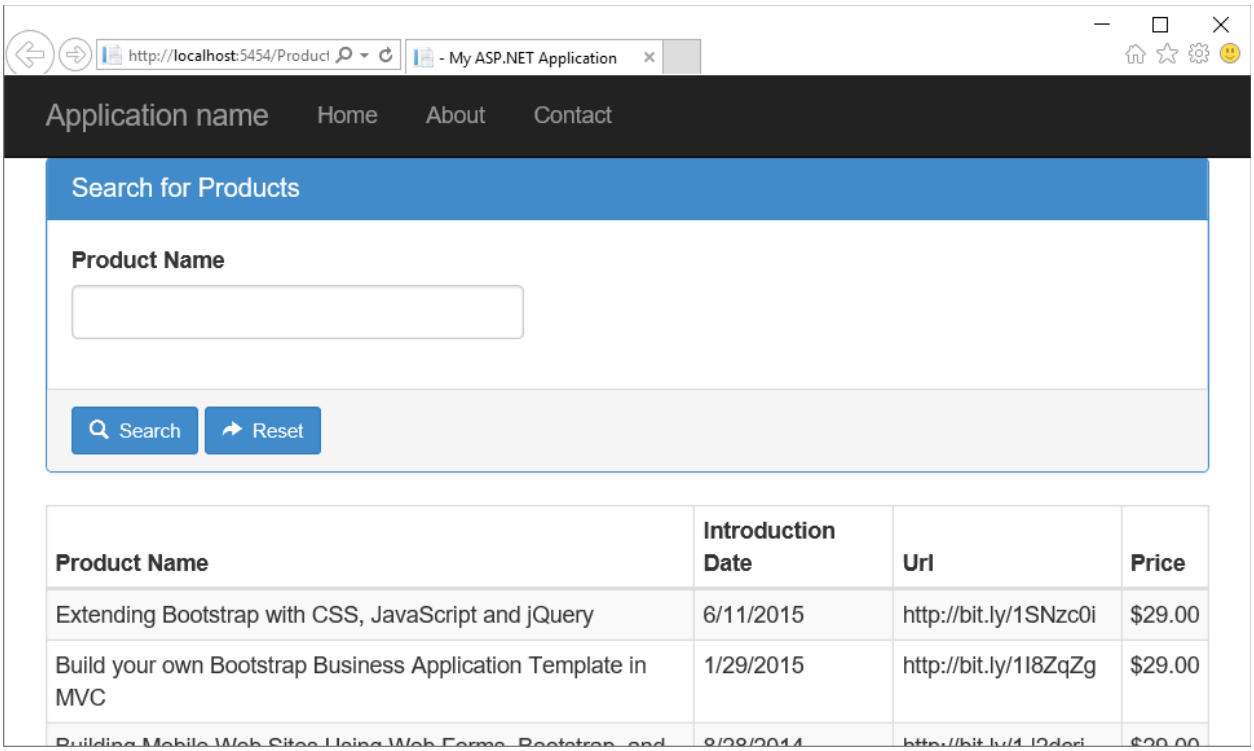

Figure 1: Add a search area for your product page

To start out, you need a class to hold the search data. Currently there is only the one search field, Product Name, but you might add additional ones later. This means you only need to add one property to this new class.

Go to PTC.DataLayer project and add a new folder called \EntityClasses. Add a new class called ProductSearch under the \EntityClasses folder. Write the following code in this class.

```
public class ProductSearch
{
  public ProductSearch() : base() {
    Init();
  }

  public void Init() {
    // Initialize all search variables
    ProductName = string.Empty;
  }

  public string ProductName { get; set; }
}
```

Go to the PTC.ViewModelLayer project and open the ProductViewModel class. Add a **using** statement at the top of this class so you can use the new ProductSearch class.

```
using PTC.DataLayer.EntityClasses;
```

Add a new property to your view model class that you can use to bind to the user interface.

```
public ProductSearch SearchEntity { get; set; }
```

Add the following line of code to the Init() method

```
SearchEntity = new ProductSearch();
```

Locate the BuildCollection method and add the following code after you set the DataCollection to the results of the db.Products.ToList().

```
// Filter the collection
if (DataCollection != null && DataCollection.Count > 0) {
  if (!string.IsNullOrEmpty(SearchEntity.ProductName)) {
    DataCollection = DataCollection.FindAll(
      p => p.ProductName
        .StartsWith(SearchEntity.ProductName,
          StringComparison.InvariantCultureIgnoreCase));
  }
}
```

Hook up the Buttons

Many MVC developers add a separate controller method for each button and hyperlink they add to a page. This leads to a lot of methods in your controller. Instead, add a string property, named EventAction, in your ProductViewModel class that tells you which button or hyperlink was pressed. This string value is going to be set into the EventAction property via a tiny bit of jQuery code. Add the following property to the ProductViewModel class.

```
public string EventAction { get; set; }
```

Modify the Init() method to initialize this property.

```
EventAction = string.Empty;
```

Go back to the PTC web project, open the Product.cshtml page, and add a <form> tag around your HTML using the Html helper BeginForm() method.

```
@using (Html.BeginForm()) {
  // HTML Code you wrote before
}
```

Add a hidden control to bind to the EventAction property you created.

```
@using (Html.BeginForm()) {
  @Html.HiddenFor(m => m.EventAction,
                  new { data_val = "false" })

  // HTML Code you wrote before
}
```

Build Search Input HTML

Add a panel to build the search area on the page. Add the following HTML below the hidden control and before the other HTML code you wrote earlier. Notice in this code you are binding to the ProductName property of the ProductSearch class you built earlier.

```
<div class="panel panel-primary">
  <div class="panel-heading">
    <h1 class="panel-title">Search for Products</h1>
  </div>
  <div class="panel-body">
    <div class="form-group">
      @Html.LabelFor(m => m.SearchEntity.ProductName,
                     "Product Name")
      @Html.TextBoxFor(m => m.SearchEntity.ProductName,
                       new { @class = "form-control" })
    </div>
  </div>
  <div class="panel-footer">
    <button id="btnSearch"
            class="btn btn-sm btn-primary"
            data-pdsa-action="search">
      <i class="glyphicon glyphicon-search"></i>
       Search
    </button>
    <button id="btnReset"
            class="btn btn-sm btn-primary"
            data-pdsa-action="resetsearch">
      <i class="glyphicon glyphicon-share-alt"></i>
       Reset
    </button>
  </div>
</div>
```

Add JavaScript File for Retrieve Action

In the HTML you just wrote, notice the **data-pdsa-action** attributes are filled in with two string values; **search** and **resetsearch**. Each time one of the buttons is clicked on, you want to take the associated string value and put it into to the hidden field. You then submit the form to have it post back to the Product controller with the SearchEntity.ProductName value from the text box filled in, and the EventAction property filled in with either “search” or “resetsearch”.

Right mouse click on the \scripts folder and select **Add | JavaScript file**. Set the name to **pdsa-action.js**. Write the following code within this file.

```
$(document).ready(function () {
  // Connect to any elements that have 'data-pdsa-action'
  $("[data-pdsa-action]").on("click", function (e) {
    e.preventDefault();

    // Fill in hidden field with action to post back to model
    $("#EventAction").val($(this).data("pdsa-action"));

    // Submit form with hidden values filled in
    $("form").submit();
  });
});
```

Go to the bottom of your Product.cshtml page and add a reference to this script file.

```
@section scripts {
  <script type="text/javascript"
          src="~/scripts/pdsa-action.js"></script>
}
```

Add Post Method in Controller

Now that you are ready to handle post backs from the user, you need to add an HttpPost method to your ProductController class. Open the ProductController class and add a new method that looks like the following.

```
[HttpPost]
public ActionResult Product(ProductViewModel vm) {

  // Handle action by user
  vm.HandleRequest();

  // Rebind controls
  ModelState.Clear();

  return View(vm);
}
```

Modify HandleRequest Method

Previously in the HandleRequest() method you had it calling the BuildCollection and nothing else. You now have a couple of new actions that can be performed. The user can choose to “search” or “resetsearch”. You need to add a switch…case to handle these different event actions. Locate the HandleRequest() method in your ProductViewModel and modify it to look like the following.

```
public void HandleRequest() {
  // Make sure we have a valid event command
  EventAction = (EventAction == null ? "" :
                  EventAction.ToLower());

  switch (EventAction) {
    case "search":
      break;

    case "resetsearch":
      SearchEntity = new ProductSearch();
      break;
  }

  BuildCollection();
}
```

Run the application, type in “b” into the Product Name search box, click the Search button, and you should see just a list of products that start with the letter “b”.

Inform User of No Rows

When the user searches and no rows are returned from that search, or if there are just no rows in the Product table, you should inform the user of this fact. You can display messages to the user using the Message property you added earlier to the ProductViewModel class. Modify the HandleRequest() method and add the following after the call to the BuildCollection() method.

```
if (DataCollection.Count == 0) {
  Message = "No Product Data Found.";
}
```

Open the Product.cshtml page and add the following lines of code around all the HTML that displays the table on this page. Don’t wrap it around the “search” area of the page.

```
@if (Model.DataCollection.Count > 0) {
  <div class="table-responsive">
  ...  // ALL THE OTHER HTML HERE
  </div>
}
else {
  <div class="row">
    <div class="col-xs-12">
      <div class="jumbotron">
        <h2>@Model.Message</h2>
      </div>
    </div>
  </div>
}
```

Run the Product page and enter a few random letters into the search text box. Click on the Search button and you should see the message displayed.

Use Partial Pages

Instead of putting all the HTML for this page on a single cshtml page, let's break each area up into separate partial pages. Copy and paste the Product.cshtml page into the \Product folder. Rename the new file to _ProductList.cshtml. Leave the @model directive at the top of the file, and then just leave the logic for building the HTML table of product data.

Once again, copy and paste the Product.cshtml page into the \Product folder. Rename the new file to _ProductSearch.cshtml. Leave the @model directive at the top of the file, and leave everything that is associated with the search area. This is the HTML between the and the for that panel.

Open the Product.cshtml and modify this file to look like the following:

```
@using PTC.ViewModelLayer
@model ProductViewModel

@using (Html.BeginForm()) {
  @Html.HiddenFor(m => m.EventAction,
                  new { data_val = "false" })

  @Html.Partial("_ProductSearch", Model)
  @Html.Partial("_ProductList", Model)
}

@section scripts {
  <script type="text/javascript"
          src="~/scripts/pdsa-action.js"></script>
}
```

Run the application and everything should still be working.

Summary

In this blog post, you broke up the MVC page into two partial pages that are called from the main page. You learned how to handle all post backs with just a single post method in your MVC controller. You created additional code in your view model to handle searching for products. In the next blog post you add a detail page and learn to add products to the product table.

Sample Code

You can download the code for this sample at www.pdsa.com/downloads. Choose the category “PDSA Blogs”, then locate the sample **Using MVVM in MVC Applications**.