

Using MVVM in MVC Applications – Part 3

This blog post continues from the last two blog posts that you can find [here](#).

- [BLOG POST 1](#)
- [BLOG POST 2](#)

In this post you add a product detail page in order to gather product data for adding to the product table. You add a save and a cancel button and learn to display validation messages. You build a method in the view model class to insert product data.

Add a Product

To add or update a product, you need to create a detail page (Figure 21) with data entry fields for each field in the table. You need to be able to display validation errors if the user does not fill out correct data for the fields. You also need to hide the Search and List areas of the screen while the user is in add or edit mode.

Create an enumeration that can help you keep track of what “mode” the page is in. There are three different modes; List, Add, and Edit. So, you need three enumerated values. Add a file in your PTC.ViewModelLayer project called PDSAPageModeEnum.cs. Remove any code in this file and replace it with the following enum definition.

```
public enum PDSAPageModeEnum
{
    List,
    Add,
    Edit
}
```

Open the ProductViewModel class and add a new property called PageMode that is of the type of this enumeration. Also, add a property called Entity, of

the type `Product`, that you can use to bind to the fields on the detail page. One more property you need is a boolean called `IsValid`. This property informs the page to display any validation errors.

```
public PDSAPageModeEnum PageMode { get; set; }
public Product Entity { get; set; }
public bool IsValid { get; set; }
```

Modify the `Init()` method to initialize these three new properties.

```
PageMode = PDSAPageModeEnum.List;
Entity = new Product();
IsValid = true;
```

Modify the `HandleRequest` method and add two new case statements. You also need to set the `PageMode` to **List** within the other case statements.

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

    Message = string.Empty;
    switch (EventAction) {
        case "add":
            IsValid = true;
            PageMode = PDSAPageModeEnum.Add;
            break;

        case "edit":
            IsValid = true;
            PageMode = PDSAPageModeEnum.Edit;
            break;

        case "search":
            PageMode = PDSAPageModeEnum.List;
            break;

        case "resetsearch":
            PageMode = PDSAPageModeEnum.List;
            SearchEntity = new ProductSearch();
            break;
    }

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

You need a way for the user to go into Add or Edit mode. Start with the Add mode by adding an Add button to the search panel as shown in Figure 20.

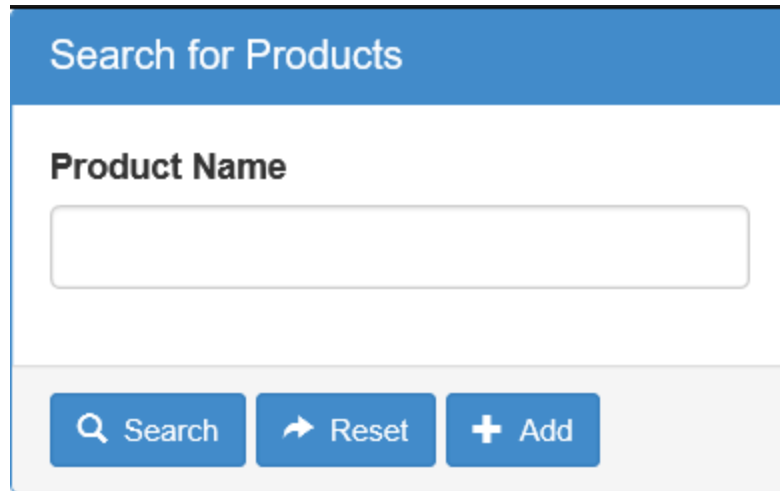


Figure 1: An Add button allows you to navigate to a blank detail page

Open the `_ProductSearch.cshtml` file and locate the Reset button. Just below this button add an addition button for the Add.

```
<button id="btnAdd"
        class="btn btn-sm btn-primary"
        data-pdsa-action="add">
  <i class="glyphicon glyphicon-plus"></i>
  &nbsp;Add
</button>
```

With just this line of code above, you can run the application and if you were to set a breakpoint on the line in the `HandleRequest` method that sets the `PageMode = PDSAPageModeEnum.Add`; you would see that you stop on that line. This is a nice feature of using the `data-pdsa-action` attribute, you simply add a new `data-pdsa-action="some value"` and the `HandleRequest` method is called with the appropriate value set in the `EventAction`. You just need to add additional case statements to handle the new actions you wish to work with.

Build the Detail Page

It is now time to build the Product data entry page. Add a new partial page in the `\Product` folder named `_ProductDetail.cshtml`. Add the following HTML to create the page shown in Figure 21.

```
@model PTC.ViewModelLayer.ProductViewModel

<div class="panel panel-primary">
  <div class="panel-heading">
    <h1 class="panel-title">
      Product Information
    </h1>
  </div>
  <div class="panel-body">
    <!-- ** BEGIN MESSAGE AREA -->
    <div class="row">
      <div class="col-xs-12">
        @if (!Model.IsValid) {
          <div class="alert alert-danger
            alert-dismissible"
            role="alert">
            <button type="button" class="close"
              data-dismiss="alert">
              <span aria-hidden="true">
                &times;
              </span>
            <span class="sr-only">Close</span>
          </button>
          <!-- Model State Errors -->
          @Html.ValidationSummary(false)
        </div>
        }
      </div>
    </div>
    <!-- ** END MESSAGE AREA -->
    <!-- ** BEGIN INPUT AREA -->
    @Html.HiddenFor(m => m.Entity.ProductId)

    <div class="form-group">
      @Html.LabelFor(m => m.Entity.ProductName,
        "Product Name")
      @Html.TextBoxFor(m => m.Entity.ProductName,
        new { @class = "form-control" })
    </div>
    <div class="form-group">
      @Html.LabelFor(m =>
        m.Entity.IntroductionDate,
        "Introduction Date")
      @Html.TextBoxFor(m =>
        m.Entity.IntroductionDate,
        new { @class = "form-control" })
    </div>
    <div class="form-group">
      @Html.LabelFor(m => m.Entity.Url, "Url")
      @Html.TextBoxFor(m => m.Entity.Url,
        new { @class = "form-control" })
    </div>
    <div class="form-group">
      @Html.LabelFor(m => m.Entity.Price, "Price")
      @Html.TextBoxFor(m => m.Entity.Price,
        new { @class = "form-control" })
    </div>
  </div>
</div>
```

```
</div>
<!-- ** END INPUT AREA -->
</div>
<div class="panel-footer">
  <div class="row">
    <div class="col-sm-12">
      <button id="btnSave"
        class="btn btn-sm btn-primary"
        data-pdsa-action="save">
        <i class="glyphicon
          glyphicon-floppy-disk"></i>
        &nbsp;Save
      </button>
      <button id="btnCancel"
        class="btn btn-sm btn-primary"
        formnovalidate="formnovalidate"
        data-pdsa-action="cancel">
        <i class="glyphicon
          glyphicon-remove-circle"></i>
        &nbsp;Cancel
      </button>
    </div>
  </div>
</div>
</div>
```

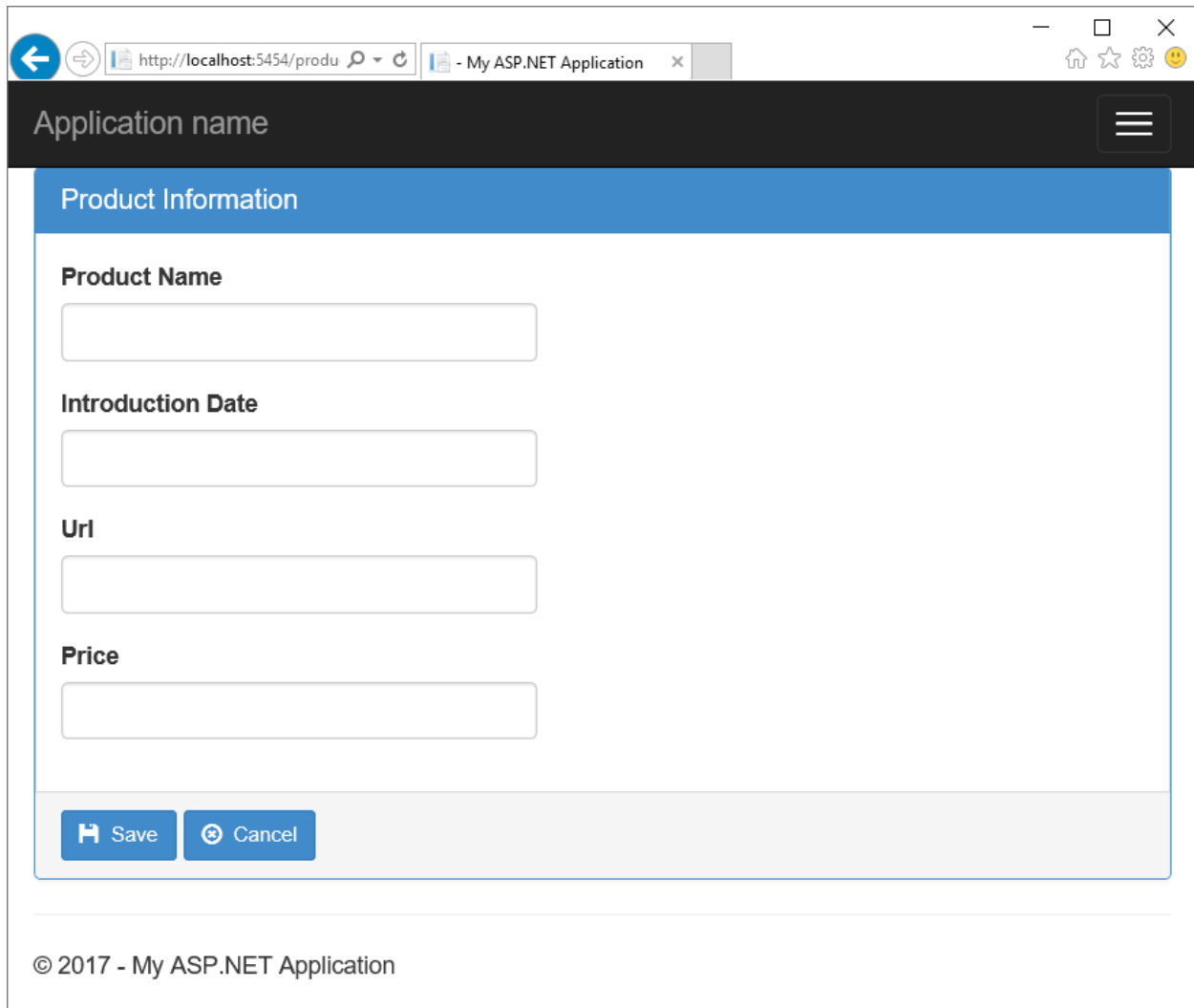
Open the Product.cshtml page and modify the code in the @using (Html.BeginForm()) area. You are going to add a new hidden field to hold the current state of the PageMode property. You are also going to wrap an **if** statement around the rendering of the partial pages. You want to display the pages based on the mode of the view model.

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

  if (Model.PageMode == PDSAPageModeEnum.List)
  {
    @Html.Partial("_ProductSearch", Model)
    @Html.Partial("_ProductList", Model)
  }

  if (Model.PageMode == PDSAPageModeEnum.Add ||
    Model.PageMode == PDSAPageModeEnum.Edit) {
    @Html.Partial("_ProductDetail", Model)
  }
}
```

You should be able to run the page, click on the Add button and you see the product detail page appear. You can't click on the Save or Cancel buttons as you have not programmed those yet.



The screenshot shows a web browser window with the URL `http://localhost:5454/produ` and a tab titled "My ASP.NET Application". The browser displays a web application with a dark header bar containing the text "Application name" and a hamburger menu icon. Below the header is a blue bar labeled "Product Information". The main content area contains a form with four input fields: "Product Name", "Introduction Date", "Url", and "Price". At the bottom of the form are two buttons: "Save" (with a floppy disk icon) and "Cancel" (with a circular arrow icon). The footer of the page reads "© 2017 - My ASP.NET Application".

Figure 2: The detail page for inserting or updating a product

Cancel Button

First create the logic in your view model class to handle the user pressing the Cancel button on the detail page. Open the `ProductViewModel` class and add a new case statement as shown below.

```
case "cancel":  
    PageMode = PDSAPageModeEnum.List;  
    break;
```

Just these three lines of code will cause the page to go back into List mode, rebuild the collection of products and display the HTML table of products. Run the application, click on the Add button, then click on the Cancel button, and you should be able to cancel out of the detail screen.

Save Button

When the user clicks on the Save button, you are going to either Insert or Update the product data depending on how the user navigated to the detail page. You have only created the Add button so far, but later you will build the edit functionality. To prepare for both inserting and updating of product data, add some method stubs to your ProductViewModel class as shown below.

```
protected void Insert() {  
  
}  
  
protected void Update() {  
  
}  
  
protected void Save() {  
    IsValid = true;  
  
    if (PageMode == PDSAPageModeEnum.Add) {  
        Insert();  
    }  
    else {  
        Update();  
    }  
}
```

Locate the HandleRequest() method and add a new case statement to handle the “save” event action.

```
case "save":  
    Save();  
    break;
```

Validation Messages

When you attempt to insert or update, the user may not put in good information and thus some validation rules may fail. You need to report those errors back to the user. Earlier you added the @Html.ValidationSummary() method to generate any validation rule failures from Data Annotations generated by the Entity Framework. However, these will only display if JavaScript is turned on in the user’s browser. And, of course, hackers can bypass these rules easily.

You also need to test business rules on the server-side in case things are bypassed, and because we can write more rules that are not able to be added as Data Annotations at the time of generation. To handle these situations, you are going to add a property to your ProductViewModel class to hold a

collection of string messages. This collection can be displayed on the `_ProductDetail.cshtml` page.

At the top of the `ProductViewModel` class add a new using statement. This statement is needed as you are going to work with some objects from the Entity Framework namespace.

```
using System.Data.Entity.Validation;
```

Create a property named `Messages` to hold a collection of messages to display to the user.

```
public List<string> Messages { get; set; }
```

Modify the `Init()` method to initialize this new collection.

```
Messages = new List<string>();
```

Add a method to add the failures contained in a `DbValidationException` object to the `Messages` collection. This exception object is thrown by the Entity Framework if any validation rules fail. You will learn how a little later. For now, write the following method.

```
protected void ValidationErrorsToMessages(
    DbEntityValidationException ex) {
    foreach (DbEntityValidationResult result in
        ex.EntityValidationErrors) {
        foreach (DbValidationError item in
            result.ValidationErrors) {
            Messages.Add(item.ErrorMessage);
        }
    }
}
```

To display these messages, or any single message, open the `_ProductDetail.cshtml` page and right below the `@Html.ValidationSummary(false)` line add the following code.

```
@if (Model.Messages.Count > 0) {
    <ul>
        @foreach (string item in Model.Messages) {
            <li>@item</li>
        }
    </ul>
}
else {
    <span>@Html.Raw(Model.Message)</span>
}
```

Insert Method

It is now time to write the code to perform the insert into the Product table. Locate the Insert() method in your ProductViewModel class and add the following code.

```
protected void Insert() {
    PTCData db = null;

    try {
        db = new PTCData();

        // Do editing here
        db.Products.Add(Entity);
        db.SaveChanges();

        PageMode = PDSAPageModeEnum.List;
    }
    catch (DbEntityValidationException ex) {
        IsValid = false;
        ValidationErrorsToMessages(ex);
    }
    catch (Exception ex) {
        Publish(ex, "Error Inserting New Product: '"
            + Entity.ProductName + "'");
    }
}
```

When the SaveChanges() method is executed, validation errors could occur here. This is when the catch block that accepts a DbEntityValidationException would execute. You then take that exception object and pass it to the ValidationErrorsToMessages method to extract the message and put it into the Messages property. These messages would then be displayed in the message area of the detail page.

Modify the Post Method in the Product Controller

Now that you are going to handle any validation errors, you want to first check to see if the ModelState property is valid. This value is automatically checked by MVC for any Data Annotations that generated validation rules to be checked. Modify the POST method in the ProductController to look like the following.

```
[HttpPost]
public ActionResult Product(ProductViewModel vm) {
    vm.IsValid = ModelState.IsValid;

    if (ModelState.IsValid) {
        // Handle action by user
        vm.HandleRequest();

        // Rebind controls
        ModelState.Clear();
    }

    return View(vm);
}
```

Run the application and click on the Add button. Immediately click on the Save button without filling any information into the page, and you should see an error message appear. Now fill in some good product information and press the Save button. You should see the new information show up in the list.

Summary

In this post, you added a product detail page to gather product data from the user. You learned how to display validation messages if the data was not input correctly. In addition, you added an insert method to the view model class in order to add data to the product table. In the next post you write code to select a single product from the product table. You also learn to update and delete product data.

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**.