Using MVVM in MVC Applications – Part 4

This blog post continues from the last two blog posts that you can find here.

- BLOG POST 1
- BLOG POST 2
- BLOG POST 3

This post is going to finish the MVC application using a MVVM approach. You are going to build the methods to select a single product from the product table. You are going to learn to update an existing product. You are also going to delete a product. Finally you learn how to handle server-side validation, and return validation messages back to the client to display to the user.

Get a Single Product

Prior to updating a record, you should go back to the table and retrieve the most current information. To do this, you require the primary key of the product record. This means you need to create a new property in your ProductViewModel class to hold this data. You also need to put the primary key value into a data- attribute, then take that value and place it into a hidden field on your page so it can be bound to the view model.

Open ProductViewModel and add a new property named EventArgument. It is in this property you are going to place the primary key value.

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

Modify the Init() method to initialize this property.

```
EventArgument = string.Empty;
```

Open the _ProductList.cshtml page

Add a new column header before all the other column headers in the <thead> area.

```
<th>Edit</th>
```

Now add a new column within the @foreach() loop.

```
<td>
  <a href="#"
     data-pdsa-action="edit"
     data-pdsa-arg="@item.ProductId"
     class="btn btn-default btn-sm">
    <i class="glyphicon glyphicon-edit"></i>
  </a>
</td>
```

Notice the new **data-pdsa-arg** attribute. This attribute's value is filled in with the primary key of the table. You need to store this value into the EventArgument property you just added to the view model class. Open the Product.cshtml page and add a new hidden field below the EventAction hidden field.

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

To add the product id to the EventArgument hidden field, open the pdsa-action.js script file and add the following line.

```
$("#EventArgument").val($(this).data("pdsa-arg"));
```

If you ran the application right now and clicked on one of the edit buttons you would see that the EventArgument property of the view model is filled in.

Write GetEntity Method

Before you update a product, it is a good idea to retrieve the complete record from the table to ensure you have the latest values. Write a GetEntity method to use the EventArgument and set the Entity property with the Product returned from the Find method of the Products collection.

```
protected virtual void GetEntity() {
  PTCData db = null;

  try {
    db = new PTCData();

    // Get the entity
    if (!string.IsNullOrEmpty(EventArgument)) {
      Entity =
       db.Products.Find(Convert.ToInt32(EventArgument));
    }
  }
  catch (Exception ex) {
    Publish(ex, "Error Retrieving Product With ID="
                 + EventArgument);
  }
}
```

Now modify the “edit” case statement in the HandleRequest to call this new method.

```
case "edit":
  IsValid = true;
  PageMode = PDSAPageModeEnum.Edit;
  GetEntity();
  break;
```

Once the Entity property is set, you display the product detail page and the data from the item selected in the list is displayed in the fields. Run the page right now and verify that you are retrieving the correct product and displaying that product in the fields on the detail page.

Update the Product

Now that you have retrieved a product from the database and displayed it on the detail page, let’s save changes back to the Product table. Add a using statement at the top of the ProductViewModel class. You need this namespace so you have access to the DbEntityValidationException class.

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

Modify the Update method in the ProductViewModel class to look like the following:

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

  try {
    db = new PTCData();

    // Do editing here
    db.Entry(Entity).State = EntityState.Modified;
    db.SaveChanges();

    PageMode = PDSAPageModeEnum.List;
  }
  catch (DbEntityValidationException ex) {
    IsValid = false;
    ValidationErrorsToMessages(ex);
  }
  catch (Exception ex) {
    Publish(ex, "Error Updating Product With ID="
                + Entity.ProductId.ToString());
  }
}
```

Run the product page and change some of the product data. Press the Save button and you should see the list refreshed with the changes you made to the one record.

Delete a Product

Now that you have added and updated a product, you should now allow the user to delete a product. To delete a product, add a button to the HTML table next to each product. Open the _ProductList.html page and add a new column header in the <thead> area.

```
<th>Delete</th>
```

Next, add a column as the last column within the @foreach() statement.

```
<td>
  <a href="#"
     data-pdsa-action="delete"
     data-pdsa-arg="@item.ProductId"
     data-pdsa-deletelabel="Product"
     class="btn btn-default btn-sm">
    <i class="glyphicon glyphicon-trash"></i>
  </a>
</td>
```

Again, you are going to use the data-pdsa-arg attribute to hold the primary key of the product record to delete. There is also one additional data- attribute called **data-pdsa-deletelabel**. This allows you to pass in the text to display when the JavaScript confirm() function is called to ask the user if they wish to "Delete this **Product**?". The value Product is what is replaced in the string. Modify the pdsa-action.js file to display a "Delete" message to the user.

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

    // Check for Delete
    if ($(this).data("pdsa-action") === "delete") {
      deletelabel = $(this).data("pdsa-deletelabel");
      if (!deletelabel) {
        deletelabel = 'Record';
      }
      if (!confirm("Delete this " + deletelabel + "?")) {
        submit = false;
      }
    }

    // Fill in hidden fields with action
    // and argument to post back to model
    $("#EventAction").val($(this).data("pdsa-action"));
    $("#EventArgument").val($(this).data("pdsa-arg"));

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

Open the ProductViewModel class and add a new method to delete a product.

```
public virtual void Delete() {
  PTCData db = null;

  try {
    db = new PTCData();

    if (!string.IsNullOrEmpty(EventArgument)) {
      Entity =
       db.Products.Find(Convert.ToInt32(EventArgument));

      db.Products.Remove(Entity);
      db.SaveChanges();

      PageMode = PDSAPageModeEnum.List;
    }
  }
  catch (Exception ex) {
    Publish(ex, "Error Deleting Product With ID="
                + Entity.ProductName);
  }
}
```

Add a new case statement in the HandleRequest for deleting a product.

```
case "delete":
  Delete();
  break;
```

Run the Product page, click on a product, answer OK when prompted, and you should see the product table refreshed with the product you deleted no longer displayed.

Add Server-Side Validation

Now that you have all the client-side validation working, add similar functionality to the server-side code as well. As it is very simple for a hacker to bypass client-side validation, you always check to ensure the data is validated on the server-side as well. To do this you add a new class to the DataLayer project named **PTCData-Extension**. After the file is added, rename the class inside of the file to **PTCData** and make it a partial class. This will allow us to add additional functionality to the PTC Entity Framework model created in part two of this blog post series.

```
public partial class PTC
{

}
```

When you attempt to insert or update data using the Entity Framework, it first calls a method named ValidateEntity to perform the validation on any data annotations added to each property. You may override this method to add your own custom validations. Add the following code to the PTC class in the PTC-Extension.cs file you just added.

```
protected override DbEntityValidationResult
  ValidateEntity(DbEntityEntry entityEntry,
  IDictionary<object, object> items) {

  return base.ValidateEntity(entityEntry, items);
}
```

Add a new method named ValidateProduct just after the ValidateEntity method you added. In this method is where you add your own custom validations. You return a list of DbValidationError objects for each validation that fails.

```
protected List<DbValidationError>
  ValidateProduct(Product entity) {
  List<DbValidationError> list =
    new List<DbValidationError>();

  return list;
}
```

The ValidateEntity method is called once for each entity class in your model that you are trying to validate. In our example, you are only validating the Product object since that is what the user is inputting. The **entityEntry** parameter passed into this method has an **Entity** property which contains the current entity being validated. Write code to check to see if that property is a Product object. If it is, pass that object to the ValidateProduct method. The ValidateProduct method returns a list of additional DbValidationError objects that need to be returned. If the list count is greater than zero, then return a new DbEntityValidationResult object by passing in the entityEntry property and your new list of DbValidationError objects.

```
protected override DbEntityValidationResult ValidateEntity(
          DbEntityEntry entityEntry,
          IDictionary<object, object> items) {
  List<DbValidationError> list =
      new List<DbValidationError>();

  if (entityEntry.Entity is Product) {
    Product entity = entityEntry.Entity as Product;

    list = ValidateProduct(entity);

    if (list.Count > 0) {
      return new DbEntityValidationResult(entityEntry, list);
    }
  }

  return base.ValidateEntity(entityEntry, items);
}
```

Now write the ValidateProduct method to perform the various validations for your Product data. Check the same validations you performed on the client-side.

```
protected List<DbValidationError> ValidateProduct(
                                        Product entity) {
  List<DbValidationError> list =
  new List<DbValidationError>();

  // Check ProductName field
  if (string.IsNullOrEmpty(entity.ProductName)) {
    list.Add(new DbValidationError("ProductName",
       "Product Name must be filled in."));
  }
  else {
    if (entity.ProductName.ToLower() ==
       entity.ProductName) {
      list.Add(new DbValidationError("ProductName",
        "Product Name must not be all lower case."));
    }
    if (entity.ProductName.Length < 4 ||
       entity.ProductName.Length > 150) {
      list.Add(new DbValidationError("ProductName",
        "Product Name must have between 4 and 150
             characters."));
    }
  }

  // Check IntroductionDate field
   if (entity.IntroductionDate < DateTime.Now.AddYears(-2)) {
    list.Add(new DbValidationError("IntroductionDate",
      "Introduction date must be within the
          last two years."));
  }

  // Check Price field
  if (entity.Price < Convert.ToDecimal(0.1) ||
     entity.Price.Value > Convert.ToDecimal(9999.99)) {
    list.Add(new DbValidationError("Price",
      "Price must be between $0.1 and
         less than $9,999.99."));
  }

  // Check Url field
  if (string.IsNullOrEmpty(entity.Url)) {
    list.Add(new DbValidationError("Url",
       "Url must be filled in."));
  }
  else {
    if (entity.Url.Length < 5 ||
       entity.Url.Length > 255) {
      list.Add(new DbValidationError("Url",
        "Url must have between 5 and 255 characters."));
    }
  }

  return list;
}
```

Run the Product page and try putting in some values that are outside the ranges specified in this method and watch the messages displayed.

Summary

In this series of blog posts you learned techniques for using the Model-View-View-Model design pattern in the context of an MVC application. There are many advantages to using MVVM in an MVC application as you saw. By using these techniques, you should find yourself writing more reusable code and code that can be tested much easier.

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**.