One-to-Many Drop-Down List in MVC Using Ajax

We often have a one-to-many relationship in tables in a database. When you need to ask the user to select a parent value, then select a child value in a web application, you don't want to post-back just to refresh the child list. Doing so causes a flash on the page and can place the user back at the top of the web page. This is not the best UI experience for the user but can be remedied easily. In this blog post, you are going to learn to populate a drop-down list based on the selection in another drop-down list. The technologies used in this post are MVC, Entity Framework, Web API, jQuery, and Ajax. To try out the samples in this blog post, create an MVC application using Visual Studio.

Data Model

For this post, I am using the sample database AdventureWorksLT for SQL Server. There is a product category table and a product table. The product table contains a foreign key reference to the product category table via a field named ProductCategoryID as shown in Figure 1.

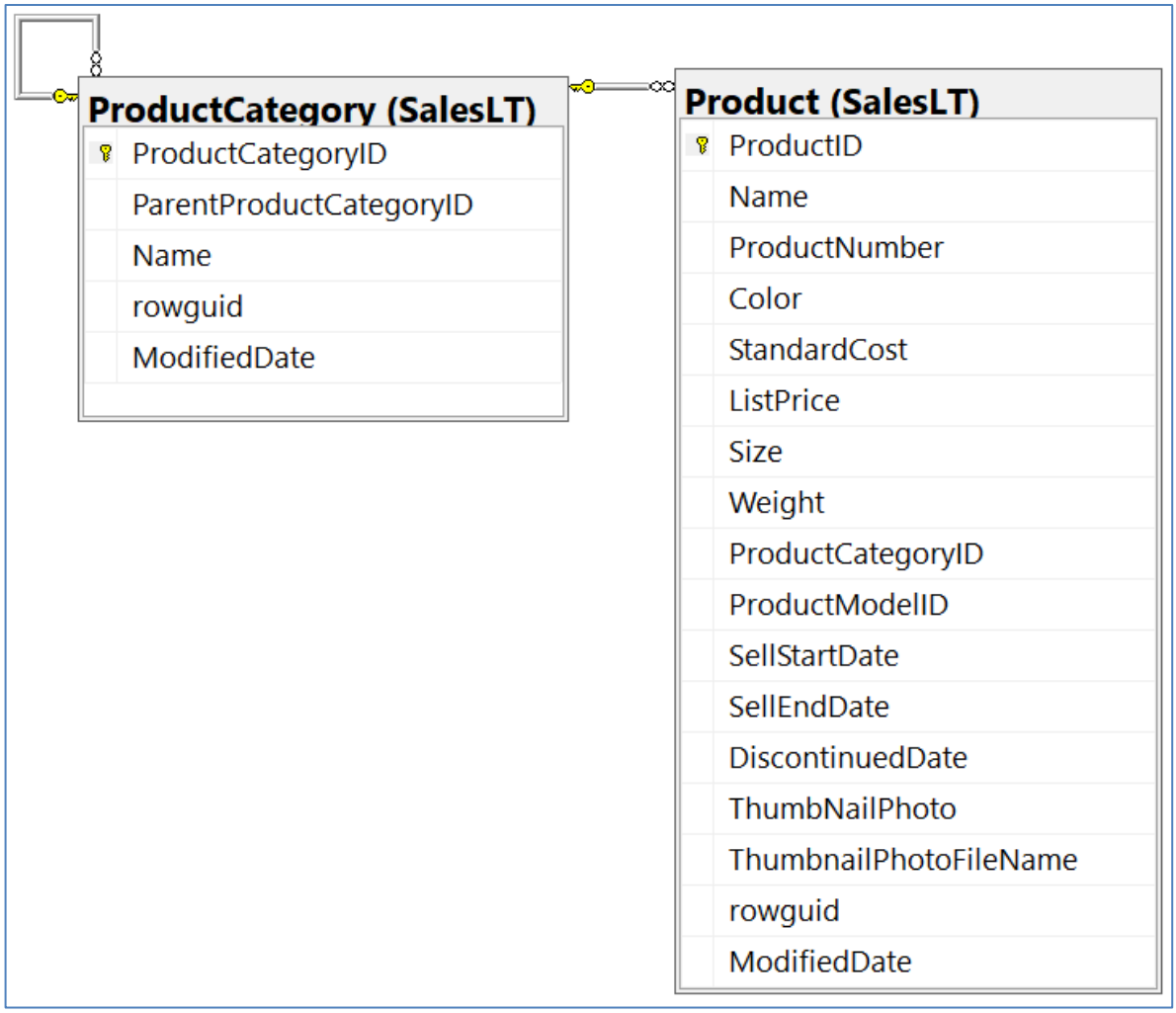

Figure 1: Relationships in the AdventureWorksLT database

Entity Classes

To access the two tables in the database, use the Entity Framework. Create two entity classes with the appropriate data annotations to connect to the two different tables. The first entity class, ProductCategory, maps to the ProductCategory table. The class definition is shown in the listing below.

```
[Table("ProductCategory", Schema = "SalesLT")]
public partial class ProductCategory
{
  [Required]
  [Key]
  public int ProductCategoryID { get; set; }
  public int? ParentProductCategoryID { get; set; }
  [Required(ErrorMessage = "Name must be filled in.")]
  public string Name { get; set; }
  public Guid? rowguid { get; set; }
  public DateTime? ModifiedDate { get; set; }
}
```

The second class, Product, maps properties to each field in the Product table.

```
[Table("Product", Schema = "SalesLT")]
public partial class Product
{
  [Required]
  [Key]
  public int? ProductID { get; set; }
  [Required(ErrorMessage = "Name must be filled in.")]
  public string Name { get; set; }
  [Required(ErrorMessage = "Product Number must be filled in.")]
  public string ProductNumber { get; set; }
  public string Color { get; set; }
  public decimal? StandardCost { get; set; }
  public decimal? ListPrice { get; set; }
  public string Size { get; set; }
  public decimal? Weight { get; set; }
  public int? ProductCategoryID { get; set; }
  public int? ProductModelID { get; set; }
  public DateTime? SellStartDate { get; set; }
  public DateTime? SellEndDate { get; set; }
  public DateTime? DiscontinuedDate { get; set; }
  public byte[] ThumbNailPhoto { get; set; }
  public string ThumbnailPhotoFileName { get; set; }
  public Guid? rowguid { get; set; }
  public DateTime? ModifiedDate { get; set; }
}
```

EF DbContext Class

A class is needed to connect to the database and provide the data connection. The class AdventureWorksLTDbContext inherits from DbContext. The DbContext class is an Entity Framework class that knows how to read and write to database tables via entity classes defined with the appropriate data annotations as shown previously.

```
public partial class AdventureWorksLTDbContext : DbContext
{
  public AdventureWorksLTDbContext() : base("name=AdventureWorksLT")
  {
  }

  public virtual DbSet<Product> Products { get; set; }
  public virtual DbSet<ProductCategory> ProductCategories
   { get; set; }
}
```

Product ViewModel

I like using a Model-View-View-Model (MVVM) approach when coding MVC applications. A ViewModel class allows me to set up all the properties needed to display on my web page. Create a class named ProductViewModel and add two properties and two methods to this class. One property holds a list of ProductCategory objects retrieved from the ProductCategory table using the LoadCategories() method. The second property, Products, holds a list of Product objects retrieved from the Product table using the LoadProductsByCategory() method.

```
public class ProductViewModel
{
  public List<ProductCategory> ProductCategories { get; set; }
  public List<Product> Products { get; set; }

  public virtual List<ProductCategory> LoadCategories()
  {
    ProductCategories = new List<ProductCategory>();

    using (AdventureWorksLTDbContext db =
             new AdventureWorksLTDbContext())
    {
      ProductCategories =
        db.ProductCategories.OrderBy(p => p.Name).ToList();
    }

    return ProductCategories;
  }

  public virtual List<Product>
     LoadProductsByCategory(int productCategoryId)
  {
    Products = new List<Product>();

    using (AdventureWorksLTDbContext db =
             new AdventureWorksLTDbContext())
    {
      Products = db.Products
        .Where(p => p.ProductCategoryID == productCategoryId)
        .OrderBy(p => p.Name).ToList();
    }

    return Products;
  }
}
```

Sample MVC Page

Create a new MVC page and add the following code. I am assuming you are using Bootstrap 3.x as your CSS framework. You are going to create two drop-down lists using the @HTML helper. The first drop-down list is loaded from the ProductCategories property. Add an onchange event to call a JavaScript function named getProducts(). The second drop-down list is going to be filled using the getProducts() function, so all you need is a simple <select> element with the *id* attribute set to "products".

```
@model DropDownAjaxSample.ProductViewModel

<div class="form-group">
  <label for="productCategories">Product Categories</label>
  @Html.DropDownList("productCategories",
    new SelectList(Model.ProductCategories,
                   "ProductCategoryId", "Name"),
    new { @class = "form-control", onchange = "getProducts();" })
</div>

<div class="form-group">
  <label for="products">Products In Category</label>
  <select class="form-control" id="products"></select>
</div>
```

Ajax Code

At the bottom of your MVC page, add a <script> tag and place the getProducts() function within the tag. This function retrieves the value selected from the first drop-down list and puts it into a variable named *catId*. An Ajax call is now created and the catId value is posted to a controller named Samples and a method named GetProductsByCategory.

When the Ajax call is completed successfully, clear the <select> element with the *id* of "products" by calling $("#products").empty(). Iterate over each element returned from the Ajax call and build an <option> element with the ProductCategoryID property value and the Name property value. Append each new option to the <select> element.

```
<script>
  function getProducts() {
    // Get Product Category ID
    var catId = $("#productCategories").val();

    // Make ajax call to get products by category
    $.ajax({
      url: "/Samples/GetProductsByCategory",
      type: 'POST',
      data: JSON.stringify({ id: catId }),
      contentType: 'application/json'
    }).done(function (response) {
      // Clear any old products
      $("#products").empty();
      if (response.length) {
        // Add new products
        $.each(response, function () {
          $('<option/>', {
            'value': this.ProductCategoryID,
            'text': this.Name
          }).appendTo('#products');
        });
      }
    }).fail(function (error) {
      console.log(error);
    });
  }
</script>
```

The Controller

Create a controller class to load the data into the ProductCategories property and to pass the ViewModel to the CSHTML page. In this controller, also add a POST method that is called from the Ajax code you wrote earlier.

```
public class SamplesController : Controller
{
  public ActionResult Sample01()
  {
    ProductCategoryViewModel vm = new ProductCategoryViewModel();

    vm.LoadCategories();

    // Add custom category to tell user to select one
    vm.ProductCategories.Insert(0,
      new ProductCategory {
        ProductCategoryID = -1,
        Name = "<-- Select a Category -->"
      });

    return View(vm);
  }

  [HttpPost]
  public JsonResult GetProductsByCategory(string id)
  {
    ProductViewModel vm = new ProductViewModel();
    List<Product> ret = new List<Product>();

    ret = vm.LoadProductsByCategory(Convert.ToInt32(id));
    if (ret.Count == 0)
    {
      ret.Add(new Product
        { ProductID = -1, Name = "No Products for Category" });
    }

    return Json(ret);
  }
}
```

The GetProductsByCategory() method returns a JsonResult object. After calling the LoadProductsByCategory() method on the ViewModel class and getting the list of products for the category selected, use the Json() method to convert the generic list of Product objects to a JsonResult object. This value is passed back to the Ajax call and allows your jQuery code to iterate over the values returned and build the list of options for the drop-down list.

Summary

In this blog post, you built two drop-down lists. One is loaded using an MVC controller, and the other is built using jQuery and a call to a Web API method. Using Ajax calls makes your application more responsive and eliminates a post back of the whole web page just to load a single drop-down list. You can build as many Web API methods to load data as you have drop-downs on your web page.

NOTE: You can download the sample code for this article by visiting my website at http://www.pdsa.com/downloads. Select “Fairway/PDSA Blog,” then select “One-to-Many Drop-Down List in MVC Using Ajax” from the dropdown list.