4.1.0 – Business Logic Layer (BLL)

Students will need the two (2) LINQ queries that are on Moodle!

Introduction

In this lesson we will use the Sequence Diagram below:

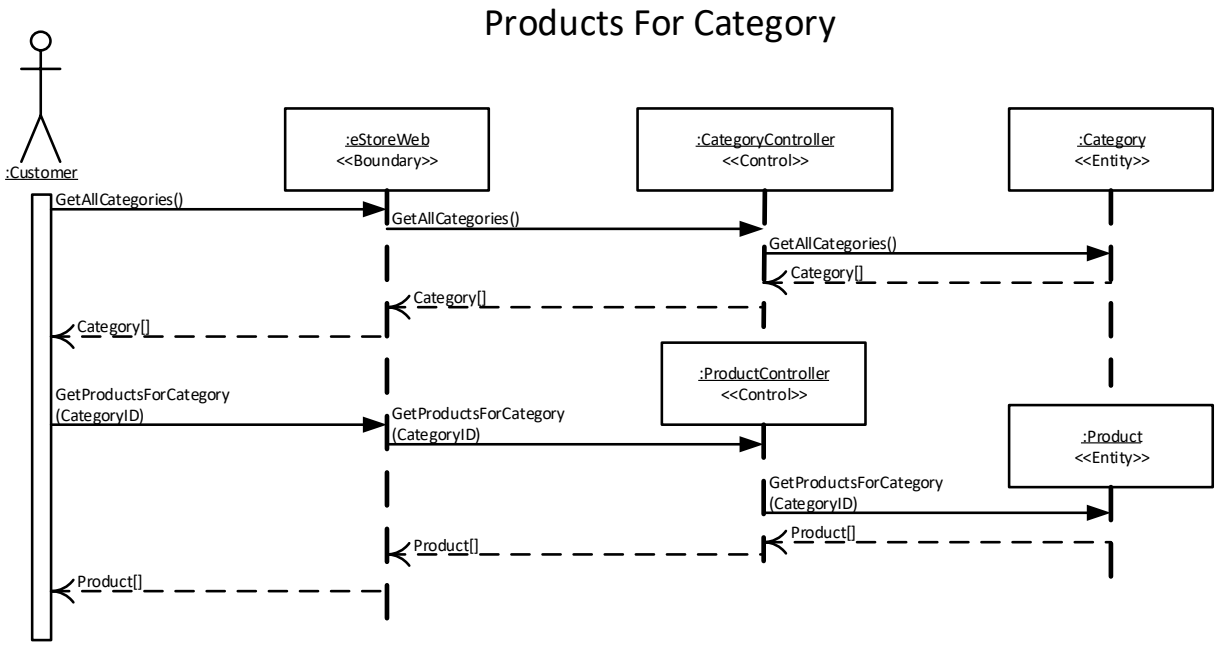

Figure 1: Products For Sale Sequence Diagram

In Lesson 3.1.0 and 3.2.0, you were shown how to create the Entity classes and add Data Annotations for validation. In this lesson, we will focus on the 2 `<<Control>>` classes. We will also need to know how to code the classes that will be in the **POCOs** folders.

Additional Data Classes

POCOs

These classes represent simple, flat, data sets. This means they are just classes with properties but do not represent any Entity class, but may look like it has properties of an Entity Class.

In the prototype screen (shown below) the data is not all the data from the Product table:

Figure 2: Prototype Web Form

It is possible to use the **`Product.cs`** class as the source of data for the GridView (or ListView), but there is a simpler way. The Products entity in LINQPad is:

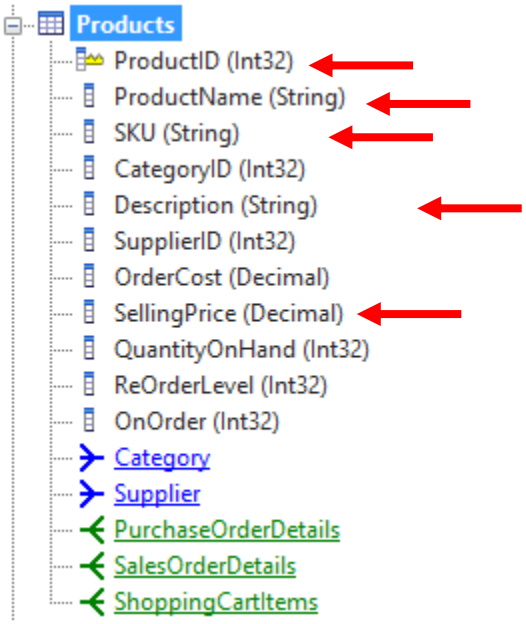

Figure 3: Products Entity in LINQPad

As we only need the 5 columns, indicated by the arrows in Figure 3 (above), it makes more sense to create a smaller class to represent the data.

In the **POCOs** folder of the **eStoreData** project add a new class called **`ProductForSale.cs`**.

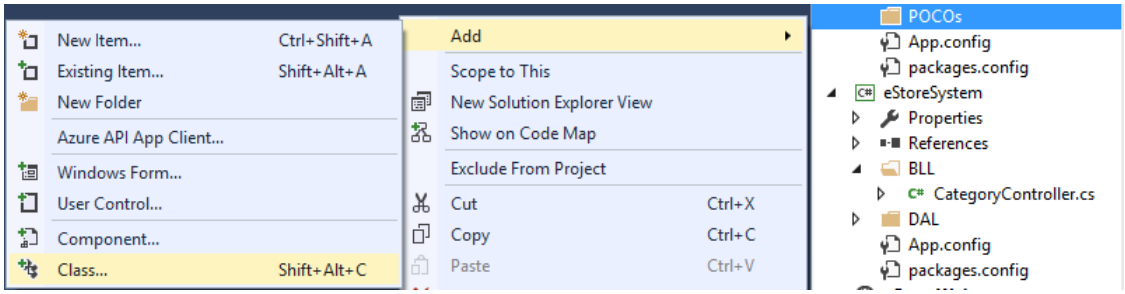

Figure 4: Add Class to POCOs Folder

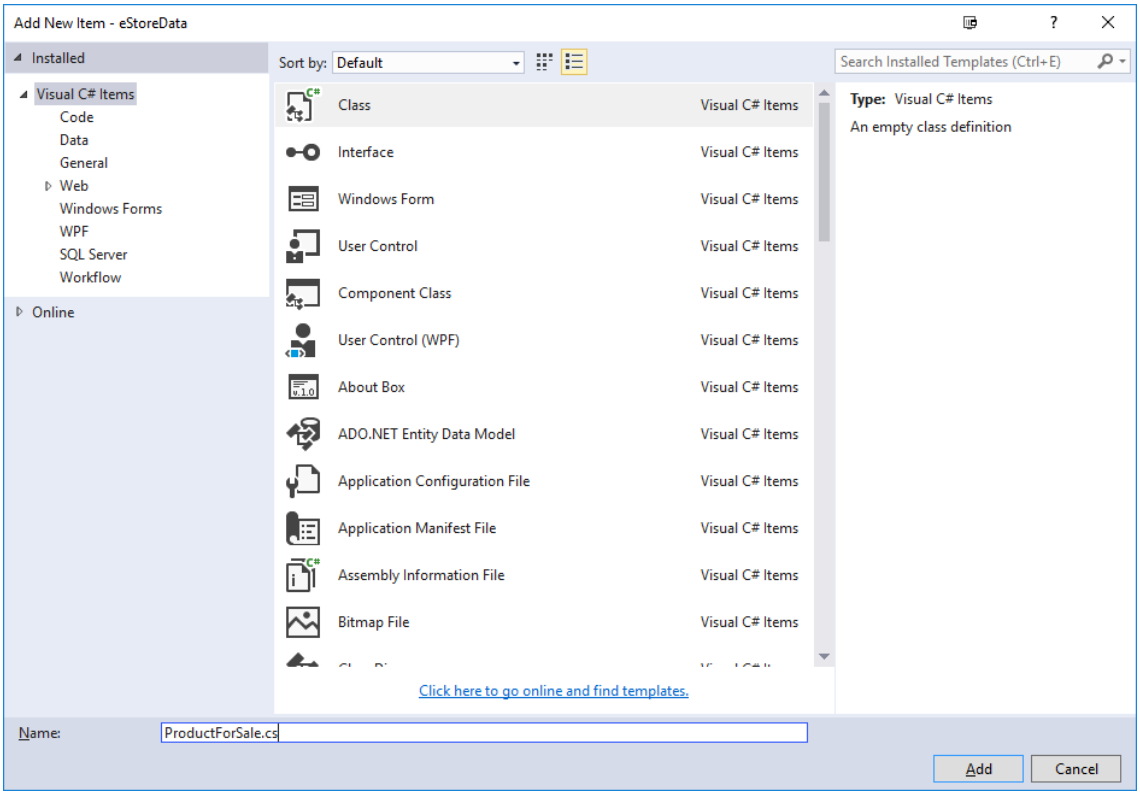

Figure 5: ProductForSale.cs

When this class is created it should look like:

Listing 1: ProductForSale.cs (Start)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace eStoreData.POCOs
{
    class ProductForSale
    {
    }
}
```

We need to:

- Make this class `public`
- Add a property for each of the columns of the table we want to use.
    - int property for ID
    - string property for Name
    - string property for SKU
    - string property for Description
    - decimal property for Price

Modify the **`ProductForSale.cs`** code to look like:

Listing 2: CategoryProduct.cs (Finish)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace eStoreData.POCOs
{
    public class ProductForSale
    {
        public int ID { get; set; }
        public string Name { get; set; }
        public string SKU { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
    }//eoc
}//eon
```

As this is our custom class, we do not need to use the same column names, and we do not need Data Annotations.

The Categories entity in LINQPad is:

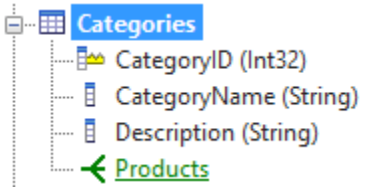

Figure 6: Categories Entity in LINQPad

On our prototype form there is a DropDownList. A DropDownList only requires 2 data properties: (1) Display property; what will be displayed on the DropDownList, and (2) Value property; what will hold the identifier, or Primary Key value. The Categories entity has 3 properties, and is has the virtual navigation property of **Products**. If we use the **`Category.cs`** class, we will get all 3 properties. It would be possible to make the DropDownList work, but a better way is to create another POCO class.

Add another class to the **POCOs** folder called **`SelectionList.cs`**. In this class modify the code to look like:

This POCO class can be reused for any data we want to bind to a DropDownList.

Remember that a DropDownList has two pieces of data: Display Text (what we see on the Dropdown), and ID Value (points to the ID or Primary Key value)

Listing 3: SelectionList.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace eStoreData.POCOs
{
    public class SelectionList
    {
        public int IDValue { get; set; }
        public string DisplayText { get; set; }
    }//eoc
}//eon
```

Business Logic Layer – Controllers

In Figure 1, we see there are 2 `<<Control>>` classes, why? In DMIT2028 you learned that on every Sequence Diagram the `<<Control>>` class **must** be the same. It is possible to code the entire solution with a single controller class, but there will be a problem. The way we access the methods of a controller class we need to create an instance of the class. When doing this, we will have access to all the public methods of that class. If all our control methods are in one class, there could be so many methods that the program would slow down. If we have separate controller classes, then the number of control methods should be small enough not to cause our code to slow down.

CategoryController

According to the Sequence Diagram, the CategoryController has only 1 method, `GetAllCategories()`. This method returns a collection, `Category[]`. First, create a new class, called **`CategoryController.cs`**, in the **BLL** folder of the **eStoreSystem** project.

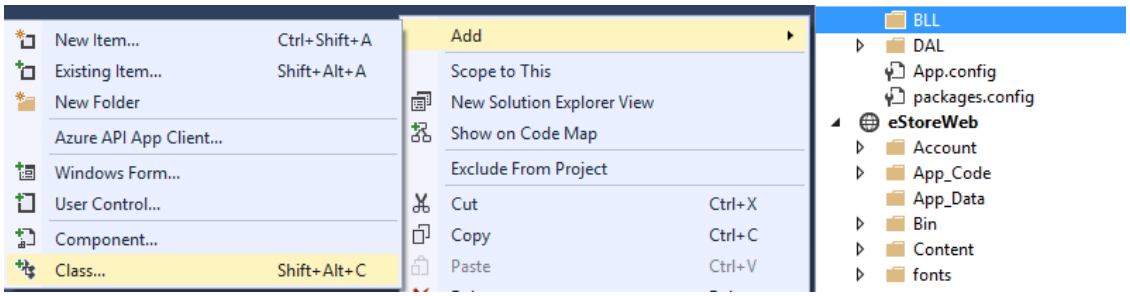

Figure 7: Add Class

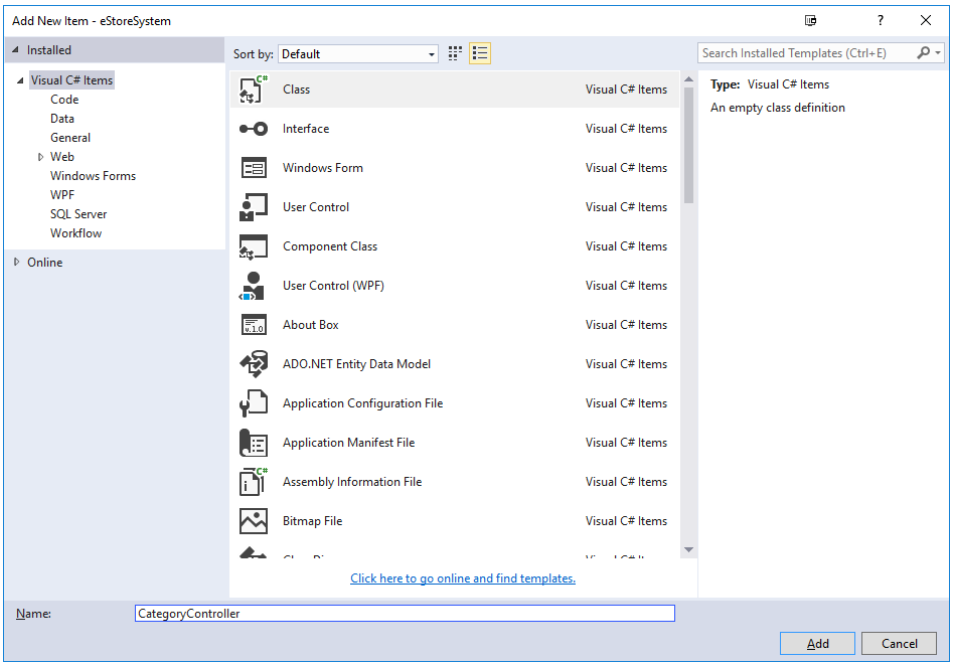

Figure 8: CategoryController

When the class file is created it should look like:

Listing 4:ProductCategoryController.cs (Start)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace eStoreSystem.BLL
{
    class CategoryController
    {
    }
}
```

Like other classes we have created, this class needs to be `public`. As this is a controller class that communicates with both the DAL and the <<Boundary>> class, it will need some additional namespaces:

Listing 5: ArtistController - Additional Namespaces

```
#region Additional Namespaces
using System.ComponentModel;
using eStoreData.Entities;
using eStoreData.POCOs;
using eStoreSystem.DAL;
#endregion
```

Needed for ObjectDataSource control.

Data Classes.

Database access.

We also need to make this class a Business Logic Layer class, or more commonly called a **Business Object**. To do that we need to annotate the class:

```
[DataObject]
public class CategoryController
```

To make the methods of this class available, as Business Object Methods, we need to annotate only those methods we want to be made available:

```
[DataObjectMethod(DataObjectMethodType.Select,false)]
public List<SelectionList> GetAllCategories()
```

Here we see the method type is a `.Select` method. This matches the UX Design document we created earlier. The `false` of the annotation means that we do not want to make this method the default method of this class. The name of the method is what we have on the Sequence Diagram. The only thing that does not match the Sequence Diagram is the type of data being returned. Instead of returning a `List<Category>` we are returning a `List<SelectionList>`. Is this a problem? No, we are only making a minor adjustment so that our code becomes more efficient.

As the method is only a READ method, we should be able to create a test query in LINQPad before we code the method in Visual Studio. In LINQPad we can create a query:

```
var results = from x in Categories
              orderby x.CategoryName
              select new
              {
                  IDValue = x.CategoryID,
                  DisplayText = x.CategoryName
              };
results.Dump();
```

Results λ SQL IL

IOrderedQueryable<> (11 items)

IDValue	DisplayText
4	Alcohol
6	Bread
7	Candy
9	Cleaners
8	Cooking
3	Drinks
1	Fruit
11	Housewares
5	Meat
10	Snacks
2	Vegetables
66	

Figure 9: LINQPad Query with Results

Once we know we can get the desired results, we can use our query in Visual Studio, with a few modifications.

When we copy and paste our query from LINQPad to Visual Studio we get some errors:

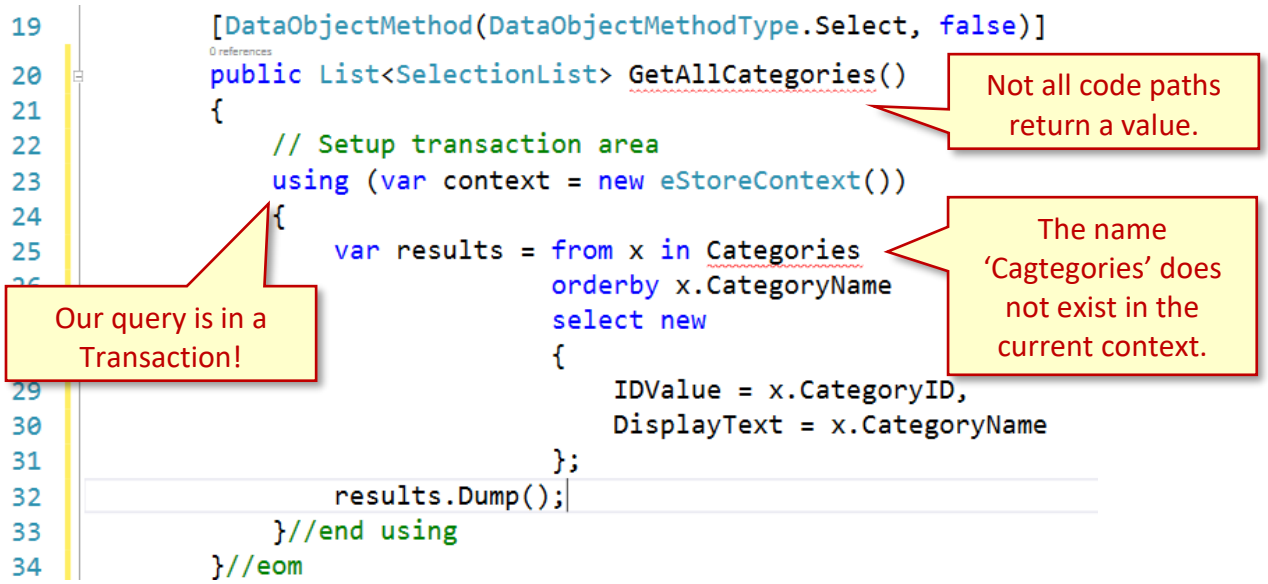

Figure 10: LINQPad Query in Visual Studio (errors)

First, we need to understand how the code should work. The line below allows us to use the DAL class, **eStoreContext.cs**, to access the data. This is done inside a Transaction (using block). The code in a Transaction will either execute with no errors and any changes will be committed to the database, or if there is one error the changes to the database will be rolled back (no changes to the database):

```
using (var context = new eStoreContext())
```

Now we need to fix the errors.

1. Change the line `var results = from x in Categories` to `var results = from x in context.Categories`.
2. Change the `select new` line to `select new SelectionList`.
3. Change the `results.Dump();` line to be `return results.ToList();`.

The complete method should look like:

Listing 6: GetAllCategories()

```
[DataObjectMethod(DataObjectMethodType.Select, false)]
public List<SelectionList> GetAllCategories()
{
    // Setup transaction area
    using (var context = new eStoreContext())
    {
        var results = from x in context.Categories
                      orderby x.CategoryName
                      select new SelectionList
                      {
```

```
                        IDValue = x.CategoryID,
                        DisplayText = x.CategoryName
                    };
        return results.ToList();
    }//end using
}//eom
```

ProductController

Using similar steps as above, create a class file called **`ProductController.cs`** in the BLL folder. Add the same additional namespaces to get:

Listing 7: ProductController.cs (Start)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

#region Additional Namespaces
using System.ComponentModel;
using eStoreData.Entities;
using eStoreData.POCOs;
using eStoreSystem.DAL;
#endregion

namespace eStoreSystem.BLL
{
    [DataObject]
    public class ProductController
    {
    }//eoc
}//eon
```

For this controller class, we will code the `GetProductsForSale(CategoryID)` message of the Sequence Diagram. Like the `GetAllCategories()` method of the CategoryController, we will need a data object method, and have a tested query from LINQPad. First, let us set up the method for this controller:

Listing 8: GetProductsForCategory(CategoryID) - Start

```
[DataObjectMethod(DataObjectMethodType.Select,false)]
public List<ProductForSale> GetProductsForCategory(int categoryID)
{
    // Setup transaction area
    using (var context = new eStoreContext())
    {

    }//end using
}//eom
```

Next, we need a tested query in LINQPad. For this query we need to hard code the CategoryID in order to get some results. The query and results should look like:

```
var results = from x in Products
              where x.CategoryID == 1
              orderby x.ProductName
              select new
              {
                  ID = x.ProductID,
                  Name = x.ProductName,
                  SKU = x.SKU,
                  Description = x.Description,
                  Price = x.SellingPrice
              };
results.Dump();
```

Results λ SQL IL

IOrderedQueryable<> (7 items)

ID	Name	SKU	Description	Price
1	Apple	F-Apple-01	Small apples	2.10
2	Apple	F-Apple-02	Medium apples	2.20
3	Apple	F-Apple-03	Large apples	2.60
4	Banana	F-Banan-01	Organic bananas	3.99
5	Banana	F-Banan-02	Imported bananas	2.38
7	Mango	F-Mango-01	Imported Orgainc Mango	3.50
6	Pineapple	F-PineA-01	Imported Orgainc Pineapple	6.25
				23.02

Listing 9: GetProductsForSale Query and Results

In this example, the CategoryID of 1 was chosen to give us several rows of data, and to match the data displayed on the prototype web form.

As before, we can copy and paste this query directly into the method, and then we will need to make some modifications. After the modifications, your code should look like:

Listing 10: GetProductsForSale(CategoryID) - Completed

```
[DataObjectMethod(DataObjectMethodType.Select, false)]
public List<ProductForSale> GetProductsForSale(int categoryID)
{
    // Setup transaction area
    using (var context = new eStoreContext())
    {
        var results = from x in context.Products
                      where x.CategoryID == categoryID
                      orderby x.ProductName
                      select new ProductForSale
                      {
                          ID = x.ProductID,
                          Name = x.ProductName,
                          SKU = x.SKU,
                          Description = x.Description,
                          Price = x.SellingPrice
                      };
        return results.ToList();
    }//end using
}//eom
```

The CategoryID of 1 was replaced with the input parameter of the method.

Build!

Build your solution! It is very important that you build your solution before you begin coding the web forms. Every time you add a class to either the **eStoreSystem** or the **eStoreData** projects you need to build your solution so that the linked library files are updated in the web site project. To see these files, you need to show all files:

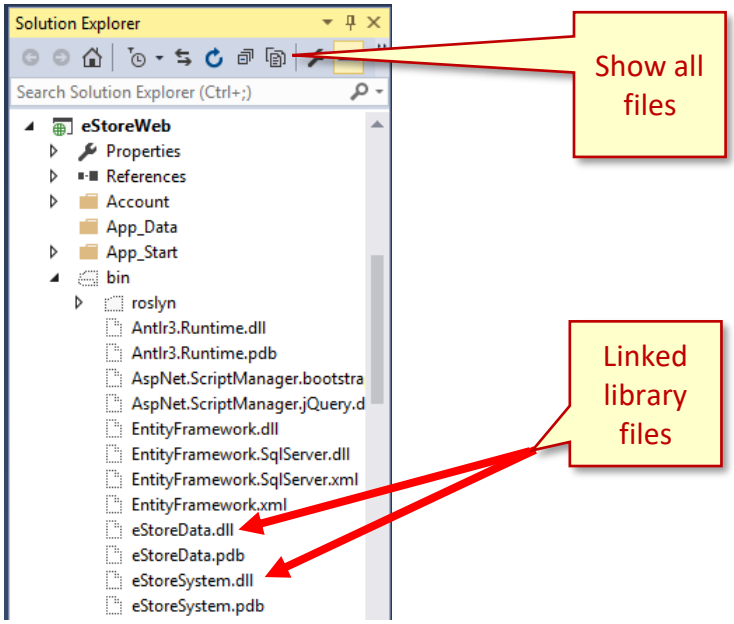

Figure 11: Bin Folder of Web Site Project

Exercise

Complete Exercise 4.1.1.