

## 8.2.0 – Transaction Demo

---

### Introduction

In previous lesson, you were introduced to the types of transactions needed for an online business. In this lesson, you will be shown how to implement an enterprise transaction; a transaction.

### Supporting Files

In the **Code Files (8.2.0)** folder on Moodle you will see the following files which will be needed for this lesson:

- CQRS\_Menu.txt: this code is to be added to the **Site.master** for new menu items
- eStoreContext\_New\_DbSets.txt: this code needs to be added to **eStoreContext.cs**
- eStoreSystem.BLL\_Contollers.zip: the files in this ZIP archive need to be extracted into your BLL folder
- eStoreSystem\_Data\_Entities.zip: the files in this ZIP archive need to be extracted into your Entities folder
- eStoreSystem\_Data\_POCOs.zip: the files in this ZIP archive need to be extracted into your POCOs folder
- SOQ.linq: the LINQ query for getting the Suggested Order Quantity (SOQ)
- Web form content code in the Shopping folder:
  - ShoppingCart\_aspx\_Content.txt
  - ShoppingCart\_aspx\_cs.txt
  - ViewCart\_aspx\_Content.txt
  - ViewCart\_aspx\_cs.txt
  - Checkout\_aspx\_Content.txt
  - Checkout\_aspx\_cs.txt
  - ViewEditPanel.txt
  - CheckoutPanel.txt
- Web form code in the Purchasing folder:
  - PurchaseOrder\_aspx\_Content.txt
  - ReceiveOrder\_aspx\_Content.txt

### Initial Setup

#### Entities Folder

Extract the files from the eStoreSystem\_Data\_Entities.zip file to your computer. From Visual Studio, right-click the Entities folder and select Add → Existing Item...:

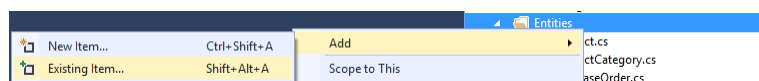


Figure 1: Add Existing Item to Entities Folder

Navigate to where you extracted the files from the ZIP file. Add all the files. Once done, you should see the following:

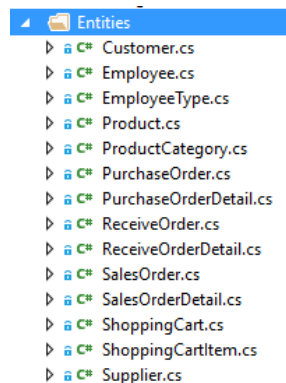


Figure 2: Entities Added

## POCOs Folder

Extract the contents of the eStore\_System\_Data\_POCOs.zip file into your POCOs folder. From Visual Studio, right-click the Entities folder and select Add → Existing Item... (use the same steps as in the Entities above).

## DAL Folder

Open the **eStoreContext.cs** file. Add the contents of the eStoreContext\_New\_DbSets.txt to replace the existing DbSet. The result should look like:

```
// Setup all DbSet Properties once Entity classes are created
public DbSet<ProductCategory> ProductCategory { get; set; }
public DbSet<Product> Product { get; set; }
public DbSet<Supplier> Supplier { get; set; }

// Added for CQRS
public DbSet<PurchaseOrder> PurchaseOrder { get; set; }
public DbSet<PurchaseOrderDetail> PurchaseOrderDetail { get; set; }
public DbSet<ShoppingCart> ShoppingCart { get; set; }
public DbSet<ShoppingCartItem> ShoppingCartItem { get; set; }
public DbSet<Customer> Customer { get; set; }
public DbSet<Employee> Employee { get; set; }
public DbSet<EmployeeType> EmployeeType { get; set; }
public DbSet<SalesOrder> SalesOrder { get; set; }
public DbSet<SalesOrderDetail> SalesOrderDetail { get; set; }
public DbSet<ReceiveOrder> ReceiveOrder { get; set; }
public DbSet<ReceiveOrderDetail> ReceiveOrderDetail { get; set; }
```

Figure 3: New DbSet Added

## BLL Folder

Select the **ProductController.cs** file and delete it. Extract the files from the eStoreSystem\_BLL\_Controllers.zip file to your computer. From Visual Studio, right-click the BLL folder and select Add → Existing Item...

Navigate to where you extracted the files from the ZIP file. Add all the files. Once done, you should see the following:

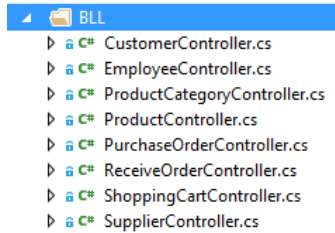


Figure 4: BLL Controllers Added

## BUILD!

Before doing anything else, make sure you **Build** your solution. There should be no errors, if there are, fix before proceeding with this lesson. Your output should look like:

```

Output
Show output from: Build
1> eStoreSystem.Data -> C:\Users\anderson\Documents\GitHub\Store_HUST\Q00_Allan_eStore\StoreSystem.Data\bin\Debug\StoreSystem.Data.dll
2>----- Build started: Project: eStoreSystem, Configuration: Debug Any CPU -----
2> eStoreSystem -> C:\Users\anderson\Documents\GitHub\Store_HUST\Q00_Allan_eStore\StoreSystem\bin\Debug\StoreSystem.dll
3>----- Build started: Project: eStoreWeb, Configuration: Debug Any CPU -----
3>Validating Web Site
3>Building directory '/App_Code/'.
3>Building directory '/Account/'.
3>Building directory '/'.
3>Building directory '/Admin/'.
3>Building directory '/UserControls/'.
3>Building directory '/Purchasing/'.
3>Building directory '/Scripts/WebForms/MSAjax/'.
3>Building directory '/Scripts/WebForms/'.
3>Building directory '/Scripts/'.
3>Building directory '/Shopping/'.
3>
3>Validation Complete
===== Build: 3 succeeded, 0 failed, 0 up-to-date, 0 skipped =====

```

Figure 5: Setup Successful Build

## Web Site Setup

### Navigation Menu

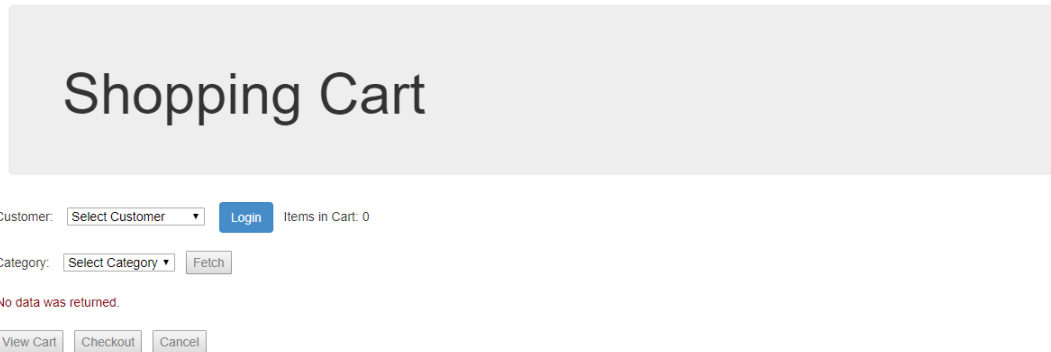
In previous lessons, you added the **Shopping** and **Purchasing** menu items. Use the contents of the CQRS\_Menu.txt file to update these menu items.

### Web Forms

1. Create the following web forms (with master) in the **Shopping** folder:
  - a. Checkout.aspx
  - b. ViewCart.aspx
2. Create the following web forms (with master) in the **Purchasing** folder
  - a. ReceiveOrder.aspx
3. Modify the content section of the following files:
  - a. **ShoppingCart.aspx** file with the code from the ShoppingCart\_aspx\_Content.txt file (replace existing code). You will have to add the MessageUserControl.
  - b. **ShoppingCart.aspx.cs** file with the code from ShoppingCart\_aspx\_cs\_txt file (replace existing code)
  - c. **PurchaseOrder.aspx** file with the code from the PurchaseOrder\_aspx\_Content.txt file (replace existing code)
  - d. **ReceiveOrder.aspx** with the code from the ReceiveOrder\_aspx\_Content.txt file

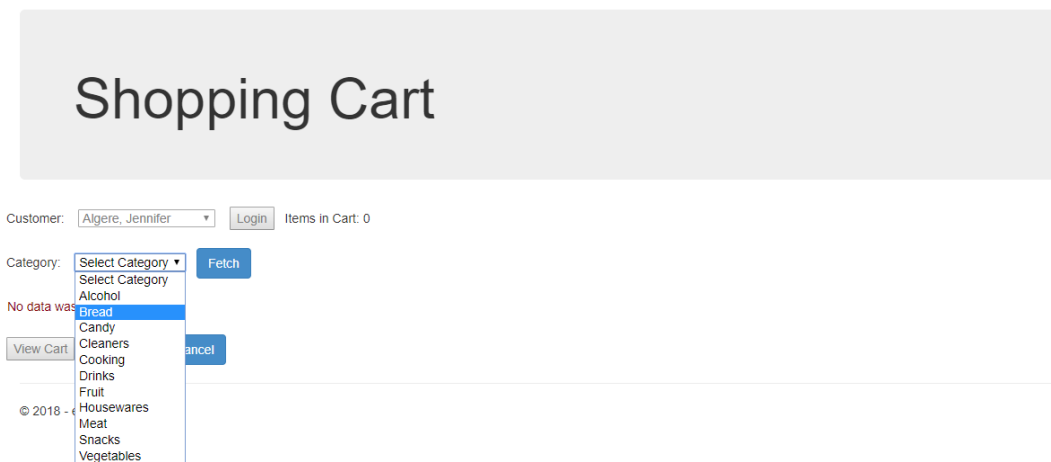
# Transaction – Shopping Cart & Sales

To truly make this an online sale, we would need the Customer to login to the web site. As we have not learned about Security, we will simulate the login with a DropDownList. The **CustomerController** has the method to return all the customers, and their ID's. Once a Customer is selected, and the **Login** button is clicked, the customer can select a category of products from a DropDownList. The customer can then add one or more products to their Shopping Cart.



The image shows a web form titled "Shopping Cart". At the top, there is a "Customer:" label followed by a dropdown menu showing "Select Customer" and a blue "Login" button. To the right of the "Login" button, it says "Items in Cart: 0". Below this, there is a "Category:" label followed by a dropdown menu showing "Select Category" and a blue "Fetch" button. Underneath the "Fetch" button, there is a red error message that says "No data was returned.". At the bottom of the form, there are three buttons: "View Cart", "Checkout", and "Cancel".

Figure 6: Initial Shopping Cart Form



The image shows the same "Shopping Cart" form as in Figure 6, but with a customer logged in. The "Customer:" dropdown now shows "Algere, Jennifer" and the "Login" button is disabled. The "Category:" dropdown is open, showing a list of categories: "Select Category", "Alcohol", "Bread", "Candy", "Cleaners", "Cooking", "Drinks", "Fruit", "Housewares", "Meat", "Snacks", and "Vegetables". The "Bread" category is highlighted. The "Fetch" button is still present. The "View Cart", "Checkout", and "Cancel" buttons are also visible. A red error message "No data was" is partially visible. At the bottom left, there is a copyright notice "© 2018 -".

Figure 7: Shopping Cart - Customer Logged in

The next step is to select a product and add it to a shopping cart (either create a new cart, or update an existing cart). In lesson 5.3.0, you learned how to select an item from a ListView. In this lesson, you will need to add the selected item to a shopping cart as well as displaying the information on the MessageUserControl and on form controls.

Added to Shopping Cart

SKU: B-Brown-01 Name: Brown Bread Quantity: 2

Customer: Algere, Jennifer Login Items in Cart: 1

Category: Bread Fetch

SKU	Name	Description	Price	Qty	
B-Brown-01	Brown Bread	Small loaf of brown bread	2.25	0	Add To Cart
B-Brown-02	Brown Bread	Large loaf of brown bread	3.10	0	Add To Cart
B-HBuns-01	Hamburger Buns	Package of hamburger buns	6.75	0	Add To Cart
B-DBuns-01	Hot Dog Buns	Package of hot dog buns	6.70	0	Add To Cart
B-TxTst-01	Texas Toast	Texas toast	4.95	0	Add To Cart
B-White-01	White Bread	Small loaf of white bread	2.35	0	Add To Cart
B-White-02	White Bread	Large loaf of white bread	3.40	0	Add To Cart

First Previous Next Last

View Cart Checkout Cancel

Only add to cart if > 0

Paging of ListView if needed

Figure 8: Sample Shopping Cart Output

When the **View Cart** button is pressed, you should get results such as:

## View/Edit Cart

	SKU	Name	Description	Price	Quantity	
Remove Edit	B-Brown-01	Brown Bread	Small loaf of brown bread	2.25	1	
Remove Edit	B-White-01	White Bread	Small loaf of white bread	2.35	1	

First Previous Next Last

Shopping Checkout Cancel

Can change Quantity or remove from cart

Figure 9: View/Edit Cart

When the **Checkout** button is pressed, you should get results such as:

## Checkout

Customer: Algere, Jennifer Sale Date: 3/19/2018 Sale Total: \$4.60

SKU	Name	Description	Price	Qty
B-Brown-01	Brown Bread	Small loaf of brown bread	2.25	1
B-White-01	White Bread	Small loaf of white bread	2.35	1

Purchase Shopping View Cart Cancel

Can create a sale or go back to add more items

Figure 10: Checkout

## BLL – ShoppingCartController.cs

Before we can complete the code for the **CreateUpdateCart** method, we need to make one modification to the **ShoppingCart** entity class. Open this file and modify it to be:

*Listing 1: Modified ShoppingCart Entity (only the namespace shown)*

```
namespace eStoreSystem.Data.Entities
{
    [Table("ShoppingCart")]
    public class ShoppingCart
    {
        // Added to allow getting ShoppingCartID
        public ShoppingCart()
        {
            ShoppingCartItems = new HashSet<ShoppingCartItem>();
        } //eom

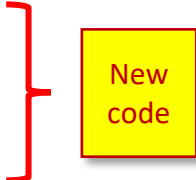
        [Key]
        public int ShoppingCartID { get; set; }

        [Required(ErrorMessage = "CustomerID is a required field.")]
        public int CustomerID { get; set; }

        [Required(ErrorMessage = "CreatedOn is a required field.")]
        public DateTime CreatedOn { get; set; }

        public DateTime? UpdatedOn { get; set; }

        // Navigation Properties
        public virtual Customer Customer { get; set; }
        public virtual ICollection<ShoppingCartItem> ShoppingCartItems { get; set; }
    } //eom
} //eon
```



The **new code** added will allow us to get the ShoppingCartID when creating a new ShoppingCart entity.

Now we can modify the **CreateUpdateCart** method to be:

*Listing 2: CreateUpdateCart Method*

```
public int CreateUpdateCart(int shoppingCartID, int customerID, int productID, int
orderQuantity)
{
    int cartID = shoppingCartID;
    ShoppingCart cart = null;
    // Setup transaction area
    using (var context = new eStoreContext())
    {
        // Is there an existing Shopping cart
        // If existing, update the UpdatedOn property
        // If not, then create a new ShoppingCart
        if(cartID == 0)
        {
            cart = new ShoppingCart();
            cart.CustomerID = customerID;
            cart.CreatedOn = DateTime.Now;
            cart.UpdatedOn = null;
        }
    }
}
```



```

        context.ShoppingCart.Add(cart);
        // need to return the new ShoppingCartID
        cartID = context.ShoppingCart.Count() + 1;
    }
    else
    {
        cart = context.ShoppingCart.Find(cartID);
        cart.UpdatedOn = DateTime.Now;
    }
    // AddToCart
    ShoppingCartItem item = new ShoppingCartItem();
    item.ShoppingCartID = cartID;
    item.ProductID = productID;
    item.Quantity = orderQuantity;
    context.ShoppingCartItem.Add(item);
    context.SaveChanges();
} //end using
return cartID;
} //eom

```

Customer's ShoppingCart exists, so just update it

Add ShoppingCartItem

In Figure 8 (above), a label displays the count of items in the Shopping Cart. The count is for this is in the **CountCartItems** method (already coded).

Figure 9 (above), shows the items in the cart. The code for this method is:

*Listing 3: ViewCart Method*

```

public List<ItemsInShoppingCart> ViewCart(int cartID)
{
    // Setup transaction area
    using (var context = new eStoreContext())
    {
        var results = from x in context.ShoppingCartItem
                       where x.ShoppingCartID == cartID
                       select new ItemsInShoppingCart
                       {
                           ItemID = x.ShoppingCartItemID,
                           ProductID = x.ProductID,
                           SKU = x.Product.ProductSKU,
                           Name = x.Product.ProductName,
                           Description = x.Product.ProductDescription,
                           Price = x.Product.SellingPrice,
                           Quantity = x.Quantity
                       };
        return results.ToList();
    } //end using
} //eom

```

This method needs the ShoppingCartID (parameter passed in to the method) that the ShoppingCartItems belong to. It uses the **ItemsInShoppingCart** POCO class.

To edit the items in the Shopping Cart, we need to code the **UpdateCart** method to be:

*Listing 4: UpdateCart Method*

```

public void UpdateCart(int cartItemID, int quantity)
{
    // Setup transaction area

```

```

using (var context = new eStoreContext())
{
    // Use a similar technique as U in CRUD
    // Update only the Quantity
    var existing = context.ShoppingCartItem.Find(cartItemID);
    existing.Quantity = quantity;
    context.Entry(existing).State = System.Data.Entity.EntityState.Modified;
    context.SaveChanges();
} //end using
} //eom

```

This method is similar to the Update method we learned in CRUD.

To remove an item from the Shopping Cart, modify the **RemoveFromCart** method to be:

```

public void RemoveFromCart(int cartItemID)
{
    // Setup transaction area
    using (var context = new eStoreContext())
    {
        // Use a similar technique as D in CRUD
        var existing = context.ShoppingCartItem.Find(cartItemID);
        context.ShoppingCartItem.Remove(existing);
        context.SaveChanges();
    } //end using
} //eom

```

As each ShoppingCartItem is unique in the database, we do not need to know which Shopping Cart the item belongs to.

## BLL – SalesOrderController

In the supplied eStoreSystem\_BLL\_Controllers.zip file you extracted to your BLL folder, there is a class file called **SalesOrderController.cs**. This file contains one method, **CreateSale**, which has 4 steps:

1. Create SalesOrder
  - OrderNumber → IDENTITY field
  - OrderDate → DateTime.Now
  - CustomerID → customerID
2. Create SalesOrderDetail & Update QuantityOnHand for each Product ordered
3. Commit changes
4. Return the new OrderNumber

The code for this method is:

*Listing 5: CreateSale*

```

public int CreateSale(int customerID, List<ItemsInShoppingCart> cart)
{
    int orderNumber = 0;
    using (var context = new eStoreContext())
    {
        // 1. Create SalesOrder

```



```

// OrderNumber --> IDENTITY field
// OrderDate --> DateTime.Now()
// CustomerID --> customerID
SalesOrder order = new SalesOrder();
order.OrderDate = DateTime.Now;
order.CustomerID = customerID;
order = context.SalesOrder.Add(order);
orderNumber = context.SalesOrder.Count() + 3000; // Sales start at 3000
// 2. Create SalesOrderDetail & update QuantityOnHand for each Product ordered
foreach (ItemsInShoppingCart item in cart)
{
    SalesOrderDetail detail = new SalesOrderDetail();
    detail.ProductID = item.ProductID;
    detail.Quantity = item.Quantity;
    detail.OrderNumber = orderNumber;
    // find the Product to update
    Product product = context.Product.Find(item.ProductID);
    // update the QuantityOnHand
    product.QuantityOnHand -= item.Quantity;
    // tell the context the Product has been modified
    context.Entry(product).State = System.Data.Entity.EntityState.Modified;
    // add the SalesOrderDetails to the SalesOrder (uses a navigation property)
    order.SalesOrderDetails.Add(detail);
} //end for
// 3. Commit changes
context.SaveChanges();
} //end using
// 5. Return the new OrderNumber
return orderNumber;
} //eom

```

**BUILD** your solution!

## Web Form – ShoppingCart.aspx

There are several techniques that can be used for shopping carts. If we had used security (the customer would login to the web site), then we could use several web pages for our shopping cart. As we are using a simulated login, the technique we will use is to have all the code on one web page. To give the user the experience of navigating several web pages, we will be using the `<asp:Panel>` control. This web control makes our web page more complex, but it allows us to have all the fields, and values needed, on one page. The only thing we have to remember is that we only make one panel visible at a time, thus the controls on the visible panel must *fit* the functions for that panel.

Take time to review the supplied web form code. Can you find all the `<asp:Panel>` controls? Also note that most of the code is already provided for you; still some work to do though.

Unlike most of the previously coded web forms, we will be writing quite a bit of code in the **ShoppingCart.aspx.cs** code behind file. The reason for this change is the **ObjectDataSource**

control is only suited for displaying data (it can be used for simple CRUD) but not for transactions such as what we will be coding. We also need to manage the `<asp:Panel>` controls, and make sure all the controls that will be visible to the user are correct.

The first thing we need to do is to add the **MessageUserController** to this web form (add it below the comment `<!-- add the MessageUserController to this div -->` line). Once added, run the page and it should look like:

Figure 11: Initial Shopping Cart Form

You should be able to select a customer from the drop down list, login using the **Login** button, select a category from the drop down list and get all the products in the selected category using the **Fetch** button. Additionally, the **Cancel** button, when clicked, returns the form to its initial view. (This was almost the same code we used in a previous lesson.)

In the div below the **MessageUserController** is the Shopping panel. On this panel is the code for the simulated login:

Listing 6: Simulated Login

```
<div class="row">
  <asp:Label ID="CustomerLabel" runat="server" Text="Customer: " /> &nbsp;&nbsp;&nbsp;
  <asp:DropDownList ID="CustomerListDDL" runat="server"
    DataSourceID="CustomerListODS"
    DataTextField="DisplayText"
    DataValueField="IDValue"
    AppendDataBoundItems="true">
    <asp:ListItem Value="0">Select Customer</asp:ListItem>
  </asp:DropDownList> &nbsp;&nbsp;&nbsp;
  <asp:Button ID="LoginButton" runat="server" Text="Login" Class="btn btn-primary"
    OnClick="LoginButton_Click" />&nbsp;&nbsp;&nbsp;
  Items in Cart:&nbsp;&nbsp;&nbsp;<asp:Label ID="CartItemCount" runat="server" Text="0" />&nbsp;&nbsp;&nbsp;
  <!-- The 2 labels below are hidden, but needed for the Shopping Cart to work -->
  <asp:Label ID="CustomerIDLabel" runat="server" Text="0" Visible="false"
/>&nbsp;&nbsp;&nbsp;
  <asp:Label ID="ShoppingCartIDLabel" runat="server" Text="0" Visible="false" />
  <br /><br />
</div>
```

In this block of code, notice the two hidden label controls. This technique is common on many web forms; we need to have these values for the form to work, but not displayed to the user. There is also a label to display the count of items in the cart.

Below this block of code is the category selection drop down list. This code is the same as we used before.

Next is the `<asp:ListView ID="ProductList_LV">`, which is already coded. Thus, entering a quantity in the Qty text box, and pressing the **Add to Cart** link button, a message will be displayed in the **MessageUserControl**. (If you do not change the value from 0, or have a negative value, an appropriate message is displayed.) Notice that attempting to add an item to the shopping cart does not update the **CartItemLabel**. We now have some work to do.

Below the closing of the first panel, you will see:

```
<asp:Panel ID="ViewEdit" runat="server" Visible="false">
    <!-- View or Edit Cart --%>
</asp:Panel>
<asp:Panel ID="Checkout" runat="server" Visible="false">
    <!-- Checkout: Create the Sale --%>
</asp:Panel>
```

(We will be adding code to these other two panels later.)

## Web Form Code Behind – ShoppingCart.aspx.cs

In the **ShoppingCart.aspx.cs** code file, the initial code is provided for you. First the code for the **Login** button:

*Listing 7: LoginButton\_Click Event Method*

```
protected void LoginButton_Click(object sender, EventArgs e)
{
    // Simulates a Customer logging in to purchase products
    int customerID = int.Parse(CustomerListDDL.SelectedValue);
    if (customerID > 0)
    {
        CustomerIDLabel.Text = customerID.ToString();
        CustomerListDDL.Enabled = false;
        LoginButton.Enabled = false;
        FetchButton.Enabled = true;
        CancelButton_SC.Enabled = true;
    }
    else
    {
        MessageUserControl.ShowInfo("LOGIN ERROR", "NO CUSTOMER SELECTED!");
    }
} //eom
```

The **Cancel** button:

*Listing 8: CancelButton\_SC\_Click Event Code*

```
protected void CancelButton_SC_Click(object sender, EventArgs e)
{
    Cancel(sender, e);
} //eom
```

This event code call the **Cancel** method (found near the bottom of this code file). As there will be a cancel button on each panel, each doing the same thing, a generic cancel method is the best way to provide this functionality:

Listing 9: Generic Cancel Method

```
protected void Cancel(object sender, EventArgs e)
{
```

```
    // Reset everything to default
    HeaderLabel.Text = "Shopping Cart";
    Shopping.Visible = true;
    ViewEdit.Visible = false;
    Checkout.Visible = false;
    ShoppingCartIDLabel.Text = "0";
    CustomerIDLabel.Text = "0";
    CustomerListDDL.Enabled = true;
    LoginButton.Enabled = true;
    FetchButton.Enabled = false;
    CancelButton_SC.Enabled = false;
    //ViewCartButton_SC.Enabled = false;
    //CheckoutButton_SC.Enabled = false;
    CartItemCount.Text = "0";
    CustomerListDDL.SelectedValue = "0";
    CategoryListDDL.SelectedValue = "0";
}
```

Turn panels on or off

Reset the labels

Reset the buttons

Uncomment these later

Reset the  
DropDownList controls

The command button on the ListView:

Listing 10: ListView Command Button Event Code

```
protected void ProductList_LV_ItemCommand(object sender, ListViewCommandEventArgs e)
{
```

```
    ListViewItem lvRow = e.Item as ListViewItem;
    int quantity = int.Parse((lvRow.FindControl("QuantityTextBox") as TextBox).Text);
    // if the user entered a quantity then process the row to add to the Shopping Cart
    if (quantity > 0)
    {
        // Get the CustomerID
        int customerID = int.Parse(CustomerIDLabel.Text);
        // Get the ShoppingCartID
        int cartID = int.Parse(ShoppingCartIDLabel.Text);
        // Get the following data from the ListView
        int productID = int.Parse((lvRow.FindControl("IDLabel") as Label).Text);
        string sku = (lvRow.FindControl("SKULabel") as Label).Text;
        string name = (lvRow.FindControl("NameLabel") as Label).Text;
        MessageUserControl.ShowInfo("Added to Shopping Cart", " SKU: " + sku + " Name: "
+ name + " Quantity: " + quantity.ToString());
        // Add to the Shopping Cart

        // Update the CartItemCount Label

        // Turn on ViewCart and Checkout buttons
    }
    else
    {
        // no quantity was entered, or the quantity < 1
        MessageUserControl.ShowInfo("ADD TO SHOPPING CART ERROR", "QUANTITY MUST BE >
0");
    }
}
```

(In this event method, we will be adding some additional code.)

The other event methods:

*Listing 11: Other Event Methods*

```
protected void ViewCartButton_SC_Click(object sender, EventArgs e)
{
    // Get the ShoppingCartID from the Label

    // Turn on only the ViewEdit Panel
} //eom

protected void CheckoutButton_SC_Click(object sender, EventArgs e)
{
    // Get the CustomerID from the Label

    // Get the ShoppingCartID from the Label

    // Turn on only the Checkout panel
} //eom
```

(In these event methods, we will be adding some additional code.)

### Add to Cart

To add an item to the cart, we need to modify the **ProductList\_LV\_ItemCommand** event method. The code we will add will provide similar functionality as an ObjectDataSource, only we will be doing this with code. The steps we will take are:

1. Add some, needed, additional namespaces
2. Create an instance of the **ShoppingCartController**
3. Call the **CreateUpdateCart** method of the controller
4. Update the **ShoppingCartIDLabel**
5. Update the **CartItemCount** label
6. Turn on the **ViewCartButton\_SC** and **CheckoutButton\_SC** buttons

Add the following between the original namespaces and the beginning of the class definition:

*Listing 12: ShoppingCart.aspx.cs - Additional Namespaces*

```
#region Additional Namespaces
using eStoreSystem.BLL;
using eStoreSystem.Data.POCOs;
using eStoreSystem.Data.Entities;
#endregion
```

Modify the existing event method to be (new lines ):

```
protected void ProductList_LV_ItemCommand(object sender, ListViewCommandEventArgs e)
{
    ListViewDataItem lvRow = e.Item as ListViewDataItem;
    int quantity = int.Parse((lvRow.FindControl("QuantityTextBox") as TextBox).Text);
    // if the user entered a quantity then process the row to add to the Shopping Cart
    if (quantity > 0)
```

```

{
    // Get the CustomerID
    int customerID = int.Parse(CustomerIDLabel.Text);
    // Get the ShoppingCartID
    int cartID = int.Parse(ShoppingCartIDLabel.Text);
    // Get the following data from the ListView
    int productID = int.Parse((lvRow.FindControl("IDLabel") as Label).Text);
    string sku = (lvRow.FindControl("SKULabel") as Label).Text;
    string name = (lvRow.FindControl("NameLabel") as Label).Text;
    MessageUserControl.ShowInfo("Added to Shopping Cart", " SKU: " + sku + " Name: "
+ name + " Quantity: " + quantity.ToString());
    // Add to the Shopping Cart
    ShoppingCartController controller = new ShoppingCartController();
    cartID = controller.CreateUpdateCart(cartID, customerID, productID, quantity);
    ShoppingCartIDLabel.Text = cartID.ToString();
    // Update the CartItemCount Label
    CartItemCount.Text = controller.CountCartItems(cartID).ToString();
    // Turn on ViewCart and Checkout buttons
    ViewCartButton_SC.Enabled = true;
    CheckoutButton_SC.Enabled = true;
}
else
{
    // No quantity was entered, or the quantity < 1
    MessageUserControl.ShowInfo("ADD TO SHOPPING CART ERROR", "QUANTITY MUST BE >
0");
}
}
} //eom

```

If you were to run this web form, it would work, but there is no code to display the contents of the shopping cart; we need to code the **ViewEdit** panel, and add appropriate code to the code behind file. First, replace the existing **ViewEdit** panel code with the contents of the ViewEditPanel.txt file. Looking at this code, you will see some of the same code used in the CRUD lesson.

Next, we need to make some code modifications to this code. For the ListView, we need to add attributes for (each of these attributes will need a new method created):

- OnItemDeleting
- OnItemEditing
- OnItemCanceling
- OnItemUPdating

Before examining the code that Visual Studio created for the four events above, we should add the OnClick events for the buttons on this panel.

```

<div class="row">
    <asp:Button ID="ShoppingButton_VC" runat="server" Text="Shopping" Class="btn btn-primary" OnClick="ShoppingButton_VC_Click"/>&nbsp;&nbsp;&nbsp;
    <asp:Button ID="CheckoutButton_VC" runat="server" Text="Checkout" Class="btn btn-primary" OnClick="CheckoutButton_VC_Click"/>&nbsp;&nbsp;&nbsp;
    <asp:Button ID="CancelButton_VC" runat="server" Text="Cancel" Class="btn btn-primary" OnClick="CancelButton_VC_Click"/>
</div>

```

## ItemsInCart\_LV\_ItemDeleting

In this event method, we need to do the following:

1. Get the item to be deleted
2. Get the parameter for removal
3. Call the **RemoveFromCart** method of the **ShoppingCartController**
4. Update the **CartItemCount** label
5. Update the List of items in the cart

Modify the event method to be:

*Listing 13: ListView – Item Deleting*

```
protected void ItemsInCart_LV_ItemDeleting(object sender, ListViewDeleteEventArgs e)
{
    // 1. Get the item to be deleted
    ListViewItem item = ItemsInCart_LV.Items[e.ItemIndex];
    // 2. Get the parameter for the removal
    int itemID = int.Parse((item.FindControl("ItemIDLabel") as Label).Text);
    // 3. Call the RemoveFromCart method of the ShoppingCartController
    ShoppingCartController controller = new ShoppingCartController();
    controller.RemoveFromCart(itemID);
    // 4. Update the CartItemCount Label
    CartItemCount.Text =
controller.CountCartItems(int.Parse(ShoppingCartIDLabel.Text)).ToString();
    // 5. Update the list of items for the cart
    ViewCartButton_SC_Click(sender, e);
} //eom
```

### ItemsInCart\_LV\_ItemEditing

In this event method, we need to do the following:

1. Check if ListView is already in Edit mode
  - a. If it is, send a message to the MessageUserControl
  - b. If not, turn editing on for the ListView

Modify the event to be:

*Listing 14: ListView – Item Editing*

```
protected void ItemsInCart_LV_ItemEditing(object sender, ListViewEditEventArgs e)
{
    // 1. Check if ListView is in Edit mode
    if (ItemsInCart_LV.EditIndex > -1)
    {
        // a. If it is, send a message to the MessageUserControl
        MessageUserControl.ShowInfo("ALERT", "View Cart is already in Edit mode.");
    }
    else
    {
        // b. If not, turn editing on for the ListView
        ItemsInCart_LV.EditIndex = e.NewEditIndex;
        ViewCartButton_SC_Click(sender, e);
    }
} //eom
```

### ItemsInCart\_LV\_ItemCanceling

In this event method, we just need to turn off editing, thus modify the event code to be:

*Listing 15: ListView - Canceling Edit*

```
protected void ItemsInCart_LV_ItemCanceling(object sender, ListViewCancelEventArgs e)
{

```

```

        ItemsInCart_LV.EditIndex = -1;
        ViewCartButton_SC_Click(sender, e);
    } //eom

```

### ItemsInCart\_LV\_ItemUpdating

In this method, we need to do the following:

1. Get the item to be updated
2. Get the parameters for the update
3. Call the **UpdateCart** method of **ShoppingCartController**
4. Update the List of items for the cart

Modify this event code to be:

*Listing 16: ListView - Update Cart*

```

protected void ItemsInCart_LV_ItemUpdating(object sender, ListViewUpdateEventArgs e)
{
    // 1. Get the item to be updated
    ListViewItem item = ItemsInCart_LV.Items[e.ItemIndex];
    // 2. Get the parameters for the update
    int itemID = int.Parse((item.FindControl("ItemIDLabel") as Label).Text);
    int productID = int.Parse((item.FindControl("ProductIDLabel") as Label).Text);
    int quantity = int.Parse((item.FindControl("QuantityTextBox") as TextBox).Text);
    // 3. Call the UpdateCart method of the ShoppingCartController
    ShoppingCartController controller = new ShoppingCartController();
    controller.UpdateCart(itemID, quantity);
    // 4. Update the list of items for the cart
    ItemsInCart_LV.EditIndex = -1;
    ViewCartButton_SC_Click(sender, e);
} //eom

```

### Updating the ListView

In each of the four event methods, there was a line of code:

```
ViewCartButton_SC_Click(sender, e);
```

This line of code calls the OnClick event code from the Shopping panel. This event now needs to be coded. The steps needed in this method (the event method stub was created, but not coded) are:

1. Get the ShoppingCartID from the Label
2. Call the **ViewCart** method of the **ShoppingCartController**
3. Bind the List to the ListView
4. Make sure the appropriate panels are visible or not visible

Modify the event method to be:

*Listing 17: Shopping ViewCart Event Method (Modified)*

```

protected void ViewCartButton_SC_Click(object sender, EventArgs e)
{
    // 1. Get the ShoppingCartID from the Label
    int cartID = int.Parse(ShoppingCartIDLabel.Text);
    // 2. Call the ViewCart method of the ShoppingCartController
    ShoppingCartController controller = new ShoppingCartController();
}

```



```

List<ItemsInShoppingCart> items = controller.ViewCart(cartID);
// 3. Bind the List to the ListView
HeaderLabel.Text = "View/Edit Cart";
ItemsInCart_LV.DataSource = items;
ItemsInCart_LV.DataBind();
// 4. Make sure the appropriate panels are visible or not visible
Shopping.Visible = false;
ViewEdit.Visible = true;
} //eom

```

## CancelButton\_VC

Like the cancel button on the Shopping panel, we just need to call the **Cancel** method. Modify the **CancelButton\_VC\_Click** event method to be:

*Listing 18: ViewEdit - Cancel Event Code*

```

protected void CancelButton_VC_Click(object sender, EventArgs e)
{
    Cancel(sender, e);
} //eom

```

## ShoppingButton\_VC\_Click

For this event method, all we need to do is turn this panel off, and turn on the Shopping panel. Modify the event method to be:

*Listing 19: ViewEdit Shopping Button Click*

```

protected void ShoppingButton_VC_Click(object sender, EventArgs e)
{
    HeaderLabel.Text = "Shopping Cart";
    Shopping.Visible = true;
    ViewEdit.Visible = false;
} //eom

```

## Test

If there are no errors in your code, you should be able to test the code so far and get results similar to Figure 9.

- Try updating the quantity of one of the items.
- Try removing an item from the cart.
- NOTE: The **Checkout** button does not work, but the **Cancel** button should work from both panels.

## Checkout

On the **Checkout** panel, we will use a GridView to display the cart items. We will also have labels for:

- Customer Name
- Sale Date
- Sale Total

There is an additional button to create the **SalesOrder** and all the **SalesOrderDetail** records. First, we need replace the existing Checkout panel code with the contents of the CheckoutPanel.txt file. (Take a few minutes to examine this code; should look familiar.) The only modifications we need to make to this code are to add the OnClick attribute for each of the buttons (create a new method for each OnClick attribute).

Three of the four OnClick events are coded below (the **CreateSaleButton** will be coded later):

*Listing 20: Ceckout - Shopping, ViewCart, and Cancel*

```
protected void ShoppingButton_CO_Click(object sender, EventArgs e)
{
    HeaderLabel.Text = "Shopping Cart";
    Checkout.Visible = false;
    Shopping.Visible = true;
} //eom

protected void ViewCartButton_CO_Click(object sender, EventArgs e)
{
    HeaderLabel.Text = "View/Edit Cart";
    Checkout.Visible = false;
    ViewEdit.Visible = true;
    ViewCartButton_SC_Click(sender, e);
} //eom

protected void CancelButton_CO_Click(object sender, EventArgs e)
{
    Cancel(sender, e);
} //eom
```

## Shopping & ViewCart Checkout Buttons

Once we have items in our cart, we need to be able to go to the **Checkout** panel from either the **Shopping** or **ViewCart** panels. Modify each of these methods as shown below:

*Listing 21: Shopping - Checkout Button*

```
protected void CheckoutButton_SC_Click(object sender, EventArgs e)
{
    HeaderLabel.Text = "Checkout";
    Shopping.Visible = false;
    Checkout.Visible = true;
    // Display the Shopping Cart
} //eom
```

*Listing 22: ViewCart - Checkout Button*

```
protected void CheckoutButton_VC_Click(object sender, EventArgs e)
{
    HeaderLabel.Text = "Checkout";
    ViewEdit.Visible = false;
    Checkout.Visible = true;
    // Display the Shopping Cart
} //eom
```

All this code does is turn on the **Checkout** panel, turn off the other panels, and set the label in the `<div class="jumbotron">`. To see the contents of the shopping cart, we could write code in both events that will do the same thing, or create a new method that does what we need, and then just call this method from the button click events. Using a new method is preferred over duplicating code.

In this new method, we need to do the following:

1. Display the Customer Name and **ShoppingCartUpdatedOn** (formatted)
2. Get the **ShoppingCartID** from the Label
3. Call the **ViewCart** method of the **ShoppingCartController**
4. Bind the List to the GridView
5. Loop through the GridView, add up all the Price \* Quantity = Total Sale, and display Total Sale on the **SaleTotalLabel**

Create a new method, after the last method of the **ShoppingCart.aspx.cs** code behind file, called **DisplayCheckout**. The code for this method is:

```
protected void DisplayCheckout(object sender, EventArgs e)
{
    // 1. Display Customer Name and ShoppingCartUpdatedOn (formatted)
    CustomerNameLabel.Text = CustomerListDDL.SelectedItem.Text;
    SaleDateLabel.Text = DateTime.Now.ToShortDateString();
    // 2. Get the ShoppingCartID from the Label
    int cartID = int.Parse(ShoppingCartIDLabel.Text);
    // 3. Call the ViewCart method of the ShoppingCartController
    ShoppingCartController controller = new ShoppingCartController();
    List<ItemsInShoppingCart> items = controller.ViewCart(cartID);
    // 4. Bind List to GridView
    SaleItems_GV.DataSource = items;
    SaleItems_GV.DataBind();
    // 5. Loop through the GridView, add up all the Price * Quantity = Total Sale
    //    and display Total Sale on the SaleTotalLabel
    decimal totalSaleCost = 0;
    foreach (GridViewRow item in SaleItems_GV.Rows)
    {
        totalSaleCost += decimal.Parse((item.Cells[5].Text)) *
int.Parse((item.Cells[6]).Text);
    }
    SaleTotalLabel.Text = String.Format("{0:c}", totalSaleCost);
} //eom
```

Now that this method is coded, we need to call it from the **Checkout** buttons of the **Shopping** and **ViewEdit** panels. Modify the two checkout buttons to be:

*Listing 23: Modified Checkout Button Events*

```
protected void CheckoutButton_SC_Click(object sender, EventArgs e)
{
    HeaderLabel.Text = "Checkout";
    Shopping.Visible = false;
    Checkout.Visible = true;
    // Display the Shopping Cart
    DisplayCheckout(sender, e);
} //eom

protected void CheckoutButton_VC_Click(object sender, EventArgs e)
{
    HeaderLabel.Text = "Checkout";
    ViewEdit.Visible = false;
    Checkout.Visible = true;
    // Display the Shopping Cart
    DisplayCheckout(sender, e);
}
```

```
}//eom
```

The last thing we do before the test is to uncomment the two commented out lines in the **Cancel()** event method.

## TEST

We need to test this and see if we can get the results shown in Figure 10. Make sure to test that you can get to the **Ckeckout** from both **Shopping** and **ViewEdit**, and that all buttons work the way we think they are to work.

## Create a Sale and Sale Details

The last thing we need to do is actually create the SalesOrder record with all the SalesOrderDetail records for the Sale. For an online sale, there is a way for the customer to pay for the sale. As financial transactions are complex, all we will be doing is creating the appropriate records in the database based on the ShoppingCart and ShoppingCartItem data we already collected on our web form.

**BUILD!** You must build your solution to be able to use the code created above. Remember to fix any errors before proceeding.

## Web Form – CreateSaleButton\_Click

When clicking the **Purchase** button, a new SalesOrder record, with its associated SalesOrderDetail records, and the QuantityOnHand for each Product sold needs to be updated. As this is a complex transaction, in which some database actions may cause an exception, we will need to handle any exception. The way to do this is to use the TryRun() method of the MessageUserController. Inside the TryRun() method, there will be 5 steps:

1. Get the CustomerID
2. Get all the ItemsInShoppingCart
3. Call the CreateSale method of the SalesOrderController
4. Display the new OrderNumber using the MessageUserController
5. Reset the web form

The code for this method is:

```
protected void CreateSaleButton_Click(object sender, EventArgs e)
{
    MessageUserController.TryRun(() =>
    {
        // 1. Get CustomerID
        int customerID = int.Parse(CustomerListDDL.SelectedValue);
        // 2. Get all the ItemsInShoppingCart
        ShoppingCartController controllerSC = new ShoppingCartController();
        List<ItemsInShoppingCart> cart =
        controllerSC.ViewCart(int.Parse(ShoppingCartIDLabel.Text));
        // 3. Call the CreateSale method of the SalesOrderController
        SalesOrderController controllerSO = new SalesOrderController();
        int orderNumber = controllerSO.CreateSale(customerID, cart);
```

```

// 4. Display the new OrderNumber using the MessageUserController
MessageUserController.ShowInfo("Sale Created", "Order Number: " + orderNumber);
// 5. Reset the web form
Cancel(sender, e);
});
} //eom

```

**NOTE:** The following steps make this work:

- 1 & 2 need to be coded first; so that we can write the code for this event method
- 3 shows that there are two controller classes; each has their own methods
- 4 calls the CreateSale; needs the parameters from the web form
- 5 will reset the form, but the MessageUserController will still display the message

## Final Test

Now that we have finished all the coding, we need to see that the data is actually adding/updating the database. Using a fresh copy of the database (restoring it to the original state), we have the following data:

### Data

	OrderNumber	OrderDate	CustomerID
1	3000	2017-04-01 14:40:02.000	1001
2	3001	2017-04-01 14:56:03.000	1003
3	3002	2017-04-01 18:18:21.000	1005
4	3003	2017-04-02 09:51:17.000	1011

Figure 12: Data – SalesOrder

	SaleOrderDetailID	OrderNumber	ProductID	Quantity
1	1	3000	1065	1
2	2	3000	1066	1
3	3	3000	1045	2
4	4	3001	1035	1
5	5	3001	1042	1
6	6	3002	1043	2
7	7	3002	1039	2
8	8	3002	1064	2
9	9	3003	1058	1
10	10	3003	1062	1
11	11	3003	1063	1
12	12	3003	1061	1
13	13	3003	1060	1

Figure 13: Data - SalesOrderDetail

	ProductID	ProductName	ProductSKU	CategoryID	ProductDescription	SupplierID	OrderCost	SellingPrice	QuantityOnHand	ReOrderLevel	OnOrder
1	1036	White Bread	B-White-01	6	Small loaf of white bread	7	1.95	2.35	20	10	0
2	1037	White Bread	B-White-02	6	Large loaf of white bread	7	2.95	3.40	20	10	0
3	1038	Brown Bread	B-Brown-01	6	Small loaf of brown bread	7	1.85	2.25	20	20	0
4	1039	Brown Bread	B-Brown-02	6	Large loaf of brown bread	7	2.85	3.10	20	30	10
5	1040	Hamburger Buns	B-HBuns-01	6	Package of hamburger buns	7	5.10	6.75	30	25	0
6	1041	Hot Dog Buns	B-DBuns-01	6	Package of hot dog buns	7	5.05	6.70	15	30	50
7	1042	Texas Toast	B-TxTst-01	6	Texas toast	7	3.25	4.95	30	25	0

Figure 14: Data - Product (Bread Category)

## Test Scenario

For this scenario:

- Customer: Algree, Jennifer (CustomerID = 1030)
- Category: Bread (CategoryID = 6)
- Items for cart:
  - B-Brown-01, Qty = 2
  - B-White-01, Qty = 3

The results:

Sale Created	
Order Number: 3004	

Figure 15: MessageUserControl.ShowInfo() Message

	OrderNumber	OrderDate	CustomerID
1	3000	2017-04-01 14:40:02.000	1001
2	3001	2017-04-01 14:56:03.000	1003
3	3002	2017-04-01 18:18:21.000	1005
4	3003	2017-04-02 09:51:17.000	1011
5	3004	2018-03-20 10:00:11.823	1030

Figure 16: SalesOrder - After Test

	SaleOrderDetailID	OrderNumber	ProductID	Quantity
1	1	3000	1065	1
2	2	3000	1066	1
3	3	3000	1045	2
4	4	3001	1035	1
5	5	3001	1042	1
6	6	3002	1043	2
7	7	3002	1039	2
8	8	3002	1064	2
9	9	3003	1058	1
10	10	3003	1062	1
11	11	3003	1063	1
12	12	3003	1061	1
13	13	3003	1060	1
14	14	3004	1038	2
15	15	3004	1036	3

Figure 17: SalesOrderDetail - After Test

	ProductID	ProductName	ProductSKU	CategoryID	ProductDescription	SupplierID	OrderCost	SellingPrice	QuantityOnHand	ReOrderLevel	OnOrder
1	1036	White Bread	B-White-01	6	Small loaf of white bread	7	1.95	2.35	17	10	0
2	1037	White Bread	B-White-02	6	Large loaf of white bread	7	2.95	3.40	20	10	0
3	1038	Brown Bread	B-Brown-01	6	Small loaf of brown bread	7	1.85	2.25	18	20	0
4	1039	Brown Bread	B-Brown-02	6	Large loaf of brown bread	7	2.85	3.10	20	30	10
5	1040	Hamburger Buns	B-HBuns-01	6	Package of hamburger buns	7	5.10	6.75	30	25	0
6	1041	Hot Dog Buns	B-DBuns-01	6	Package of hot dog buns	7	5.05	6.70	15	30	50
7	1042	Texas Toast	B-TxTst-01	6	Texas toast	7	3.25	4.95	30	25	0

Figure 18: Product (Bread Category) - After Test