

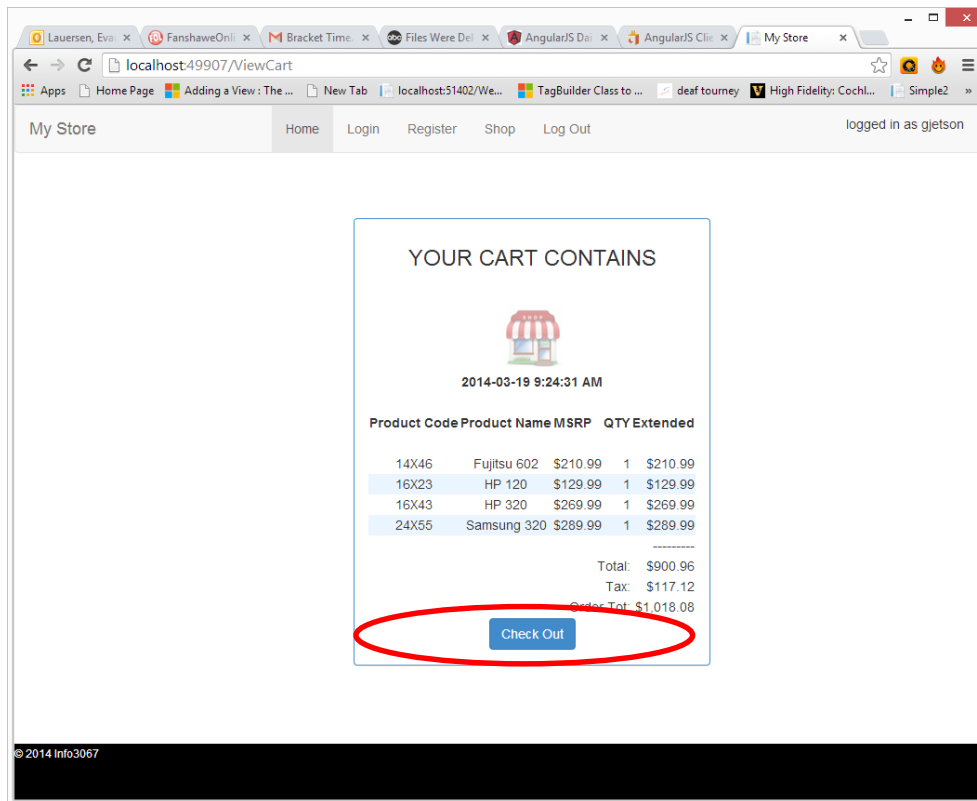
INFO3067 Week 4 Class 2

Review

- ViewCart

Case Study cont'd – Checkout

- We're now ready to check out



Transactions

- Remembering “The Happy Ice Cream Seller”
 - 4 possible outcomes
 - 2 acceptable
- The role of the DBMS
 - Commit v rollback

Preparing the Database for Orders

The database basically has 4 entities to control the order process, they are the 4 tables and their corresponding entities (shown in parenthesis):

1. Orders (Order) – table containing order information
2. OrderLineItems (OrderLineItem) – table containing the detailed line item information
3. Products (Product) – table containing inventory amounts for products
4. Customers (Customer) – table for containing customer information

Two of the tables we have already used (Customers and Products). In our original SQL we made sure the Orders table had an identity property set on its primary key (OrderID),





To add the orders we'll need a new class in our eStoreModels project called **OrderModel.cs**. The process for adding an order is pretty much identical to adding one for a Customer so it won't be repeated here. There is one minor difference, because we are adding both orders and line items we need to encompass both types of adds in a **transaction**. First you will need to add a .Net reference to the eStoreModels project for **System.Transaction** and before you set up the Try/Catch for order/orderlineitem processing, insert the following transaction scope **using** to house our transaction:

```
// Define a transaction scope for the operations.
using (TransactionScope transaction = new TransactionScope())
{
    try
    {
        dbContext = new eStoreDBEntities();
        Order myOrder = new Order();

        // back to the db to get the right Customer based on session customer id
        var selectedCusts = dbContext.Customers.Where(c => c.CustomerID == cid);
        myOrder.Customer = selectedCusts.First();

        // ...
    }
}
```

Looking at our Orders table in the database we see we need the following fields

Columns
 OrderID (PK, int, not null)
 OrderDate (datetime, not null)
 OrderAmount (money, not null)
 CustomerID (FK, int, not null)

So we see need to retrieve only the OrderAmount and CustomerID from a passed dictionary, we can use the current time for the OrderDate value. Also notice in the code above, the framework wants an actual Customer Entity not just the CustomerID field.

Now adding the line items is a bit more involved because we are dealing with products that make up our existing inventory. As we add line items to the order we must take the stock out of circulation. We'll also need to handle the scenario of what happens when we run out of current stock. Typically a company will still bill the customer and create what's called a **backorder**, which is kind of like an I.O.U. to the customer indicating that they will ship the goods when the next batch arrives. In the following code see how both scenarios are handled:

```
dbContext.Orders.Add(myOrder); // Add Order and get OrderID for Line Item

for (int idx = 0; idx < qty.Length; idx++)
{
    if (qty[idx] > 0)
    {
        OrderLineitem item = new OrderLineitem();
        string pcd = prodcd[idx];
        var selectedProds = dbContext.Products.Where(p => p.ProductID == pcd);
        item.Product = selectedProds.First(); // got product for item

        if (item.Product.QtyOnHand > qty[idx]) // enough stock
        {
            item.Product.QtyOnHand = item.Product.QtyOnHand - qty[idx];
            item.QtySold = qty[idx];
            item.QtyOrdered = qty[idx];
            item.QtyBackOrdered = 0;
            item.SellingPrice = sellPrice[idx];
        }
        else // not enough stock
        {
            item.QtyBackOrdered = qty[idx] - item.Product.QtyOnHand;
            item.Product.QtyOnBackOrder = item.Product.QtyOnBackOrder + (qty[idx] - item.Product.QtyOnHand);
            item.Product.QtyOnHand = 0;
            item.QtyOrdered = qty[idx];
            item.QtySold = item.QtyOrdered - item.QtyBackOrdered;
            item.SellingPrice = sellPrice[idx];
            boFlg = true; // something backordered
        }
        myOrder.OrderLineitems.Add(item);
    }
}
```

Also notice that there are two changes made to the database an update to the product table in addition to the insert into the line item table. Couple those with the insert into the order table and **we have a total of 3 tables affected** by the new order, hence the **need for a transaction**. We want all 3 changes to be made or none of them to be made. The transaction will look after problems as any exception will cause a rollback to automatically occur.

This code uses 3 arrays to process the transaction (qty[], prodcd[], and sellPrice[]). Where does the data for these 3 arrays come from? Well the ViewModels layer will need to pass it down from the web page. It doesn't matter how you pass the data just

make sure it's packaged in a serialized dictionary. Here is how I received the data in the OrderModel object:

```
/// <returns>populated dictionary with new order #, bo flag or error</returns>
public byte[] AddOrder(byte[] bytOrder)
{
    Dictionary<string, Object> dictionaryOrder = (Dictionary<string, Object>)Deserializer(bytOrder);
    Dictionary<string, Object> dictionaryReturnValues = new Dictionary<string, Object>();
    // deserialize dictionary contents into local variables
    int[] qty = (int[])dictionaryOrder["qty"];
    string[] prodcd = (string[])dictionaryOrder["prodcd"];
    Decimal[] sellPrice = (Decimal[])dictionaryOrder["msrp"];
    int cid = Convert.ToInt32(dictionaryOrder["cid"]);

    bool boFlg = false;
    // ... (rest of the code) ...
}
```

Lastly after the order and all line items have been added don't forget to actually save all the changes, see the code below (also notice you can test to see if the transaction works with the commented code).

In summary, we loop through the qty array (that comes from the ViewModels layer) and add a line item for each item that has a qty greater than zero. We need to keep track of the backorder status so we set the "boflag" variable to 1 if any of the goods are backordered and return it and the generated Order# back to the ViewModels layer (again I don't care how you do this, I used a dictionary). If there is any exception we return the error message. I used an element called "message" in the dictionary.

```
        myOrder.OrderLineitems.Add(item);
    }
}

dbContext.SaveChanges(); // made it this far, persist changes
// throw new Exception("Rollback"); // test trans by uncommenting out this line

// Mark the transaction as complete.
transaction.Complete();

dictionaryReturnValues.Add("orderid", myOrder.OrderID);
dictionaryReturnValues.Add("boflag", boFlg);
dictionaryReturnValues.Add("message", "");
}
catch (Exception e) // if the catch is hit, the trans will be rolled back by the framework
{
    ErrorRoutine(e, "OrderData", "AddOrder");
    dictionaryReturnValues.Add("message", "Problem with Order");
}
return Serializer(dictionaryReturnValues);
}
}
```

OrderViewModel.cs

This ViewModel object will basically package the order information up and send it on to the Model layer (via serialized dictionary). From the code below you can see where the various pieces of information are coming from. In return it will receive the generated order # and an indication if any of the goods were on back ordered if successful or an error message if the order was not generated.

```
/// <returns></returns>
public void AddOrder(CartItem[] items, int cid, double amt)
{
    Dictionary<string, Object> dictionaryReturnValues = new Dictionary<string, Object>();
    Dictionary<string, Object> dictionaryOrder = new Dictionary<string, Object>();

    try
    {
        Message = "";
        BackOrderFlag = 0;
        OrderID = -1;
        OrderModel myData = new OrderModel();
        int idx = 0;
        string[] prodcds = new string[items.Length];
        int[] qty = new int[items.Length];
        Decimal[] sellPrice = new Decimal[items.Length];

        foreach (CartItem item in items)
        {
            prodcds[idx] = item.ProdCd;
            sellPrice[idx] = item.Msrp;
            qty[idx++] = item.Qty;
        }

        dictionaryOrder["prodcd"] = prodcds;
        dictionaryOrder["qty"] = qty;
        dictionaryOrder["msrp"] = sellPrice;
        dictionaryOrder["cid"] = cid;
        dictionaryOrder["amt"] = amt;
        dictionaryReturnValues = (Dictionary<string, Object>)Deserializer(myData.AddOrder(Serializer(dictionaryOrder)));
        OrderID = Convert.ToInt32(dictionaryReturnValues["orderid"]);
        BackOrderFlag = Convert.ToInt32(dictionaryReturnValues["boflag"]);
        Message = Convert.ToString(dictionaryReturnValues["message"]);
    }
    catch (Exception ex)
    {}
}
```

Lastly all you need to do is code the ViewCartController's **genOrder** method which is triggered by the button click event in the View to start the process:

```

/// <summary>
/// Add the order, pass cart, customer # and order amount to ViewModel
/// </summary>
/// <returns></returns>
public ActionResult genOrder()
{
    OrderViewModel myOrder = new OrderViewModel();
    try
    {
        myOrder.AddOrder((CartItem[])Session["Cart"],
                        Convert.ToInt32(Session["CustomerID"]),
                        (double)Session["OrderAmt"]);


        if (myOrder.OrderID > 0) // Order Added
        {
            ViewBag.Message = "Order " + myOrder.OrderID + " Created!";
            if (myOrder.BackOrderFlag > 0)
                ViewBag.Message += " Some goods were backordered!";
        }
        else // problem
            ViewBag.Message = myOrder.Message + ", try again later!";
    }

    catch (Exception ex)
    {
        ViewBag.Message = "Order was not created, try again later! - " + ex.Message;
    }
    return PartialView("PopupMessage");
}

```

YOUR CART CONTAINS

Order 2 Created! Some goods were backordered!



2014-03-19 9:44:52 AM

Product Code	Product Name	MSRP	QTY	Extended
14X46	Fujitsu 602	\$210.99	1	\$210.99
14X48	Fujitsu 2650	\$329.99	2	\$659.98
16X23	HP 120	\$129.99	1	\$129.99
16X43	HP 320	\$269.99	1	\$269.99

				Total: \$1,270.95
				Tax: \$165.22
				Order Tot: \$1,436.17

Notice that once the order is created, the button is hidden so that the user cannot press it again and generate double orders. To accomplish this, the form for this View can be laid out as follows, notice the **OnComplete** section:

```

@using (
    Ajax.BeginForm("genOrder", "ViewCart", new AjaxOptions
    {
        LoadingElementId = "ajaxSplash",
        InsertionMode = InsertionMode.Replace,
        HttpMethod = "GET",
        UpdateTargetId = "messg2",
        OnComplete = "$('#orderButton').hide()"
    })
{

```

Lastly confirm that the Order and OrderLineItem data have the correct data in each of the tables:

SQLQuery1.sql - EV...Evan-PC\Evan (53))* X

```

SELECT * FROM Orders WHERE OrderID =2
SELECT * FROM OrderLineItems WHERE OrderID=2

```

100 %

Results Messages

	OrderID	OrderDate	OrderAmount	CustomerID	Timer
1	2	2014-03-19 09:45:05.407	1270.95	7	0x000000000000007F3

	LineItemID	OrderID	ProductID	QtyOrdered	QtySold	QtyBackOrdered	SellingPrice	Timer
1	5	2	14X46	1	1	0	210.99	0x000000000000007F4
2	6	2	14X48	2	0	2	329.99	0x000000000000007F5
3	7	2	16X23	1	1	0	129.99	0x000000000000007F6
4	8	2	16X43	1	1	0	269.99	0x000000000000007F7

You can find included in this week's content area, the formal requirements in a file called **Case1Requirements.pdf**. There is also a list of terms to study for the upcoming midterm exam, that document is called **MidtermStudy.pdf**.

The midterm review will occur next class (Week 5 Class 1) followed by the Case study 1 demonstration (case is due at the **start** of the 2nd hour). The midterm exam is scheduled for class 2 next (Week 5 Class 2) in the first hour, and we'll dive into the 2nd part of the case study in the last 2 hours.