

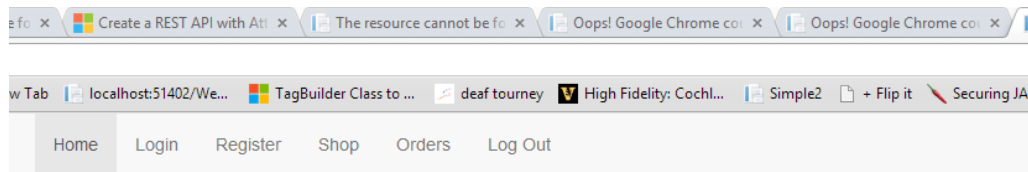
INFO3067 Week 6 Class 1


Review

- SMTP processing
- JQuery 1-6

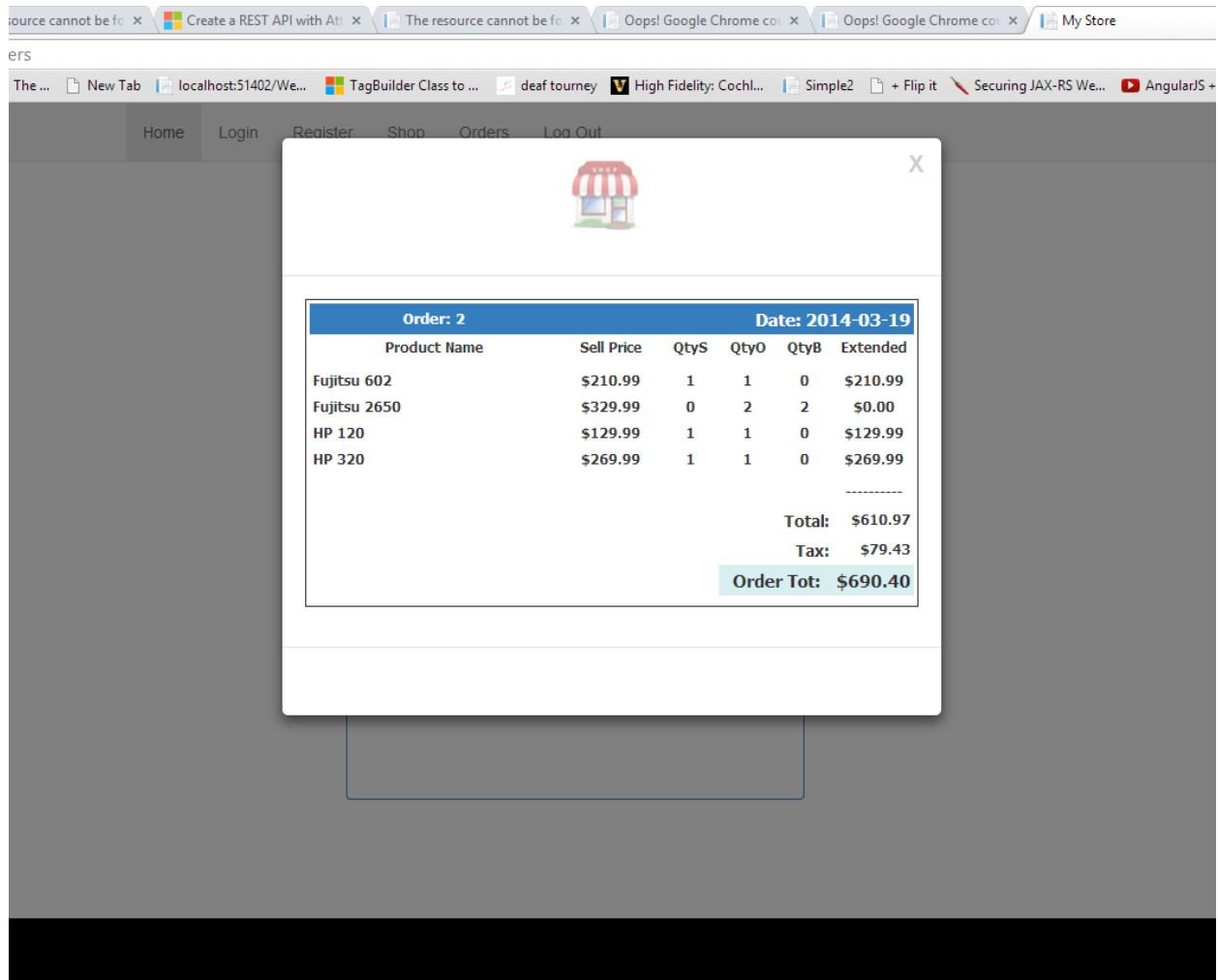
Using Client Side JQuery to View Orders

- We are going to provide a list of orders that this particular customer has purchased. To allow the customer to view the details of the order they simply need to click on one of the order #'s.:



YOUR ORDERS	
	
ID	Date
1	2014-03-19
2	2014-03-19

- For example if the user clicked on the 2 under the ID column they would get the details for order #2 – see next page.



- You'll notice that the process is actually quite similar to the JQuery 5/6 exercise from last class. As you recall the process is started by creating a click event handler in our javascript:

```
$("td").click(function() {
```

- As a safe guard you need to ensure that the user has clicked on the right table, as theoretically a user could click on a different page's table cell, so throw an if in this event handler:

```
if ($(this).attr('id') == 'OrderNo')
```

- There are actually two parts to this page:
 - Generate the list (1st screen shot above)
 - Generate JSON based order information that the JQuery code will process and format the details popup (2nd screen shot above).

Producing the Initial Order List

When the user chooses the Orders menu item, we'll pass control over to a new **OrdersController**. The **Index** method in turn will return the default view. The default view should employ an html helper to build the initial list. The helper needs to obtain all order #'s and dates that the current customer has placed to date, and then build the markup with the data embedded.

OrderController.Index→View→Helper←→OrderViewModel←→OrderModel

The pattern is one we have seen before so I am not providing any details for this process.

Once the helper has built the list we then need to modify the td.click function from the JQuery exercises a bit, to obtain the number from the cell. We don't need any elaborate string manipulation but rather a simple call like:

```
orderId = $(this).text();
```

We don't want to keep going back to the server for each order that the user wants to view the details for, rather we'll bring all of the order information back for this user in one call. Then we can process any subsequent viewing entirely client side. To obtain the data for all orders for this user we need to grab the data from not only the orders table **but also** the OrderLineItems and Products table. There are a number of ways to do this, instead of building a stored procedure with a joined query and setting up a Function call as we did in INFO3070, we can also do the join using LINQ, either method is fine.

Starting in the OrdersModel class, create a method that returns a list of a new container class (I called it OrderDetailsModel), you can obtain the structure of this container class from the method's code below:

```

/// <summary>
/// Get All details for all orders for a customer
/// </summary>
/// <param name="custid"> customerid</param>
/// <returns>List of details for all orders customer owns </returns>
public List<OrderDetailsModel> GetAllDetailsForAllOrders(int custid)
{
    List<OrderDetailsModel> allDetails = new List<OrderDetailsModel>();

    try
    {
        eStoreDBEntities db = new eStoreDBEntities();

        // LINQ way of doing INNER JOINS
        var results = from o in db.Orders
                      join l in db.OrderLineItems on o.OrderID equals l.OrderID
                      join p in db.Products on l.ProductID equals p.ProductID
                      where (o.CustomerID == custid)
                      select new OrderDetailsModel
                      {
                          OrderID = o.OrderID,
                          ProductName = p.ProductName,
                          SellingPrice = l.SellingPrice,
                          QtySold = l.QtySold,
                          QtyOrdered = l.QtyOrdered,
                          QtyBackOrdered = l.QtyBackOrdered,
                          OrderDate = o.OrderDate
                      };

        allDetails = results.ToList<OrderDetailsModel>();
    }
    catch (Exception e)
    {
        ErrorRoutine(e, "OrderData", "GetAllDetailsForAllOrders");
    }
    return allDetails;
}

```

Moving up the chain, the OrderViewModel class will have to have a method that receives this list and converts it to a list that the View can be aware of. I just created a container/dto (data transfer object) class with exactly the same properties as OrderDetailsModel and called it **OrderDetailsViewModel**.

The List class contains a method called **ConvertAll** I used it to copy the List coming from the Models layer to a list that we can send to the view. Note you can use the traditional method of employing a for..each loop to do the same thing and again either are fine:

```

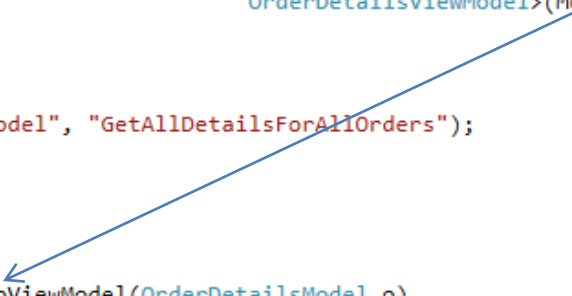
/// </summary>
/// <returns>List of OrderDetailWeb instances to Presentation layer</returns>
2 references
public List<OrderDetailsViewModel> GetAllDetailsForAllOrders()
{
    List<OrderDetailsViewModel> viewModelDetails = new List<OrderDetailsViewModel>();

    try
    {
        OrderModel myData = new OrderModel();
        List<OrderDetailsModel> modelDetails = myData.GetAllDetailsForAllOrders(CustomerID);

        // this could be done with a foreach loop as well - see above
        viewModelDetails = modelDetails.ConvertAll(new Converter<OrderDetailsModel,
                                                    OrderDetailsViewModel>(ModelToViewModel));
    }
    catch (Exception ex)
    {
        ErrorRoutine(ex, "OrderViewModel", "GetAllDetailsForAllOrders");
    }
    return viewModelDetails;
}

1 reference
private OrderDetailsViewModel ModelToViewModel(OrderDetailsModel o)
{
    OrderDetailsViewModel v = new OrderDetailsViewModel();
    v.OrderID = o.OrderID;
    v.ProductName = o.ProductName;
    v.QtyBackOrdered = o.QtyBackOrdered;
    v.QtyOrdered = o.QtyOrdered;
    v.QtySold = o.QtySold;
    v.SellingPrice = o.SellingPrice;
    v.OrderDate = o.OrderDate;
    return v;
}
}

```

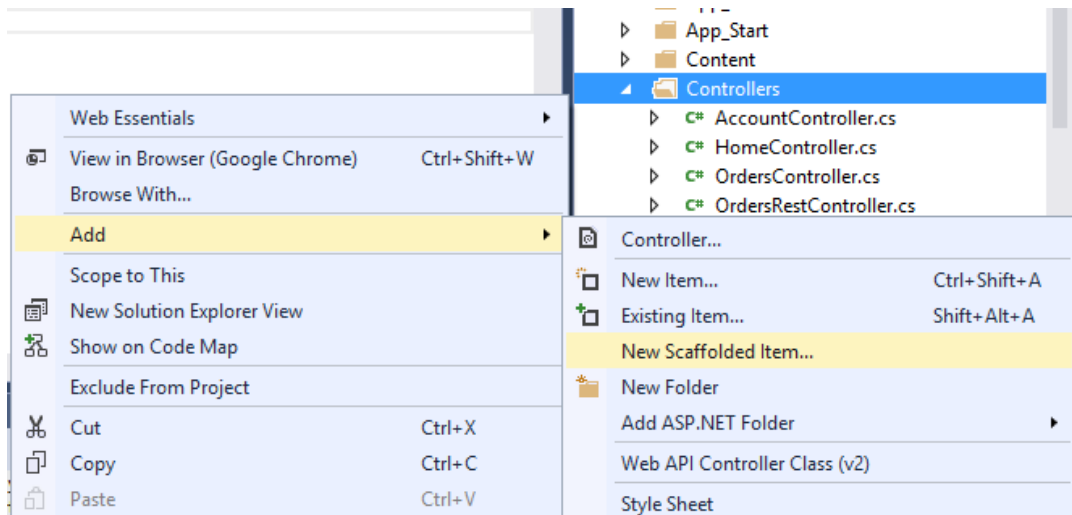


Documentation for the **ConvertAll** method can be found here:

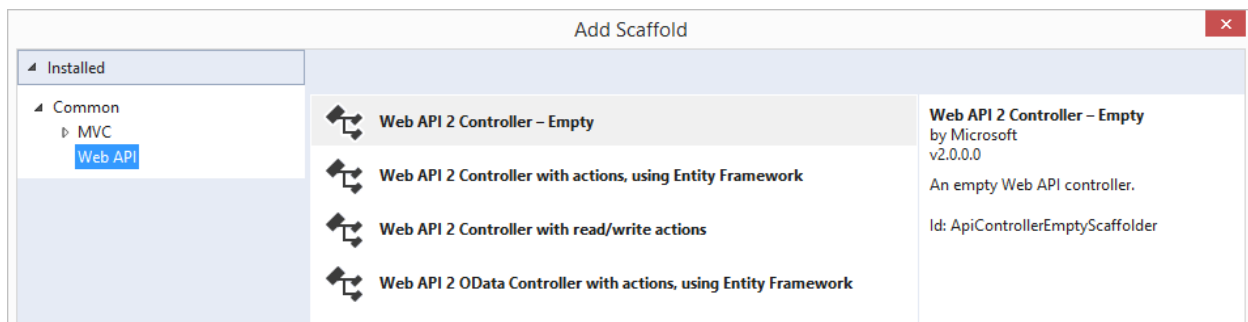
<http://msdn.microsoft.com/en-us/library/73fe8cwf.aspx>

Now with our desired information available to the View, we just need to get it into JSON format. Fortunately there is an easy way to do this. We'll define a new type of controller using Microsoft's latest web technology called **Web API**. A Webapi controller class creates what's called a REST service. You'll study REST services in depth in 6th semester. For now just think of a rest service as an easy way to ask the server for JSON data.

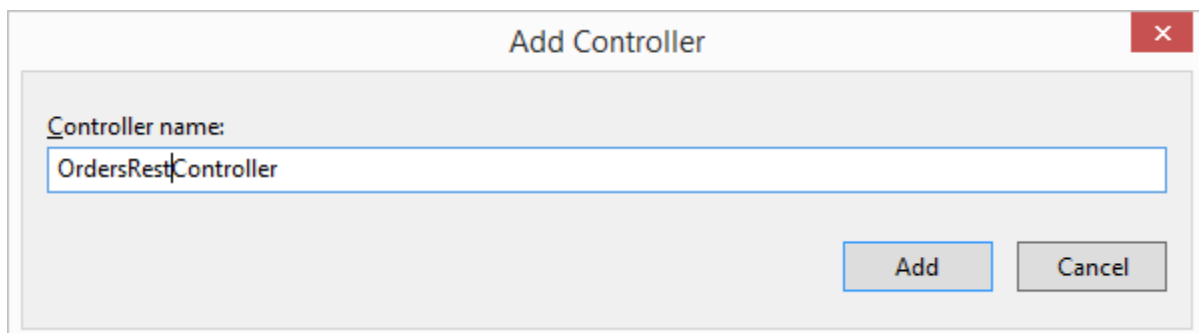
Add a new scaffolded item to the Controllers folder as follows:



And then choose the Web API section and the **Web API 2.0 Controller – Empty** option:



Give it a name of **OrdersRestController**:



The wizard may return the following output:

```
TestRestController.cs Web.config JQuery1.htm JQuery2.htm JQuery3.htm JQuery4
Visual Studio has added the full set of dependencies for ASP.NET Web API 2 to project 'MVCExercises'.

The Global.asax.cs file in the project may require additional changes to enable ASP.NET Web API.

1. Add the following namespace references:

    using System.Web.Http;
    using System.Web.Routing;

2. If the code does not already define an Application_Start method, add the following method:

    protected void Application_Start()
    {
    }

3. Add the following lines to the beginning of the Application_Start method:

    GlobalConfiguration.Configure(WebApiConfig.Register);
```

Locate the **Global.asax** file in your web site project and add the suggested GlobalConfiguration.Configure line plus one more that will force the service to just return JSON (no xml) .in the Application_Start method. Note you will need to add the System.Web.Http using as well:

```
MvcExercises.MvcApplication Application_Start()
using System.Web.Mvc;
using System.Web.Routing;
using System.Web.Http;

namespace MVCExercises
{
    References
    public class MvcApplication : System.Web.HttpApplication
    {
        References
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            GlobalConfiguration.Configure(WebApiConfig.Register);
            GlobalConfiguration.Configuration.Formatters.XmlFormatter.SupportedMediaTypes.Clear(); // remove xml form
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

We can return a .json result by adding a method back in our new OrdersRestController and call that method directly from the JQuery routine to pull the data down to the client: in the correct format:

```

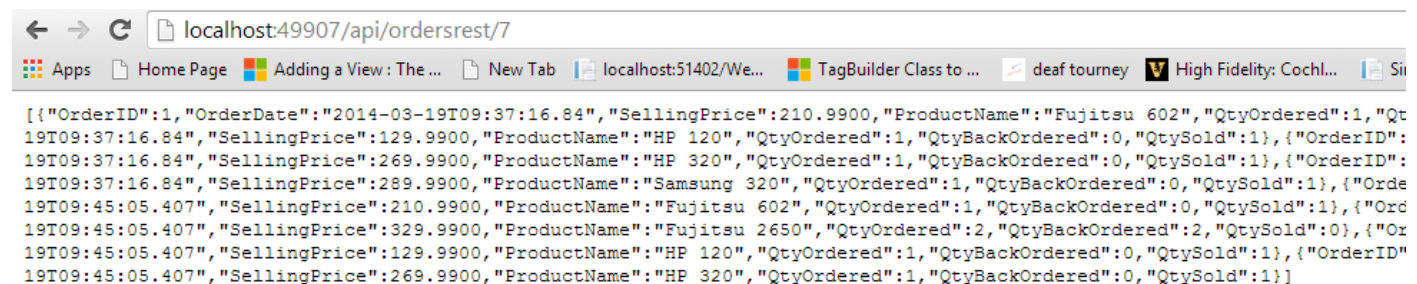
using System.Web.Mvc;
using eStoreViewModels;

namespace eStoreWebsite.Controllers
{
    // References
    public class OrdersRestController : ApiController
    {
        // GET api/ordersrest
        // References
        public IHttpActionResult Get(int id)
        {
            OrderViewModel myOrder = new OrderViewModel();
            myOrder.CustomerID = id;
            List<OrderDetailViewModel> orders = myOrder.GetAllDetailsForAllOrders();
            return Ok(orders); //http 200
        }
    }
}

```

Before we test this out with JQuery insure the method works by calling directly in the browser. Start the website, and login with a known user, then use the following URL (note you'll need to get the customer id of the signed on user):

<http://localhost:#####/api/ordersrest/#> (##### = port number, # = customer id)



```

[{"OrderID":1,"OrderDate":"2014-03-19T09:37:16.84","SellingPrice":210.9900,"ProductName":"Fujitsu 602","QtyOrdered":1,"QtyBackOrdered":0,"QtySold":1}, {"OrderID":19T09:37:16.84","SellingPrice":129.9900,"ProductName":"HP 120","QtyOrdered":1,"QtyBackOrdered":0,"QtySold":1}, {"OrderID":19T09:37:16.84","SellingPrice":269.9900,"ProductName":"HP 320","QtyOrdered":1,"QtyBackOrdered":0,"QtySold":1}, {"OrderID":19T09:37:16.84","SellingPrice":289.9900,"ProductName":"Samsung 320","QtyOrdered":1,"QtyBackOrdered":0,"QtySold":1}, {"OrderID":19T09:45:05.407","SellingPrice":210.9900,"ProductName":"Fujitsu 602","QtyOrdered":1,"QtyBackOrdered":0,"QtySold":1}, {"OrderID":19T09:45:05.407","SellingPrice":329.9900,"ProductName":"Fujitsu 2650","QtyOrdered":2,"QtyBackOrdered":2,"QtySold":0}, {"OrderID":19T09:45:05.407","SellingPrice":129.9900,"ProductName":"HP 120","QtyOrdered":1,"QtyBackOrdered":0,"QtySold":1}, {"OrderID":19T09:45:05.407","SellingPrice":269.9900,"ProductName":"HP 320","QtyOrdered":1,"QtyBackOrdered":0,"QtySold":1}]

```

Once you have confirmed the method is returning the correct JSON formatted data we can call this method directly from the client using JQuery's getJSON method, Note In the code below the variable **allOrders** is predefined:


```

// getOrders
// - obtain the order# from click event, call server
// - for JSON array using WEBAPI for all orders for this customer (hidden field)
// - pass array, order# and order date to build routine
function getOrders(oid, allOrders) {
    if (allOrders == null) { // only go to the server once

        $.getJSON("api/ordersrest/" + $("#customer").val(), null, function (data, status, jqXHR) {
            buildOrder(oid, orderDate, data); // call is asynchronous, so build in if
            allOrders = data; // before loading global data for else
        }).error(function (jqXHR, textStatus, errorThrown) { //problem
            console.log(textStatus + " - " + errorThrown);
        });
    }
    else {
        buildOrder(oid, orderDate, allOrders);
    }
} // getOrders

```

The parameters for this call are:

- "api/ordersrest" – Defaults to Controller/method on the host with a GET/id
- function(data,status,jqXHR) – if call was successful run this inline function with the returned JSON formatted **data**, the **status** of the call (will be success if everything is ok, and a **JQuery XMLHttpRequest** object that you can use for further debugging)
- further explanation of the \$.getJSON call can be found here: <http://api.jquery.com/jquery.getJSON/>

The allOrders/**data** variable is identical in function to the hardcoded array we had in the JQuery exercises. Then it's just a matter of using the data and building the order in a similar fashion like we did with JQuery exercises, except we only want the data for the desired order, so we'd code something like the following in the buildOrder function to get the order we need:

```

$.each(data, function (index, order) {
    if (order.OrderID == oid) { // make sure we
        tr = $("<tr>");
        td = $("<td>");
    }
}

```

The \$.each function here loops through the data variable (a json array in our case) and each entry in data is known by the variable order, further info on this statement can be found here: <http://api.jquery.com/jquery.each/>

Lab 9

- Submit 2 screen shots selecting 2 different orders (see page 1)
- Submit the JQuery code you used to create the rendered order (in the code above it would be my `getOrders` and `buildOrder` functions)