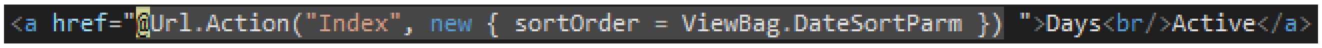
<u>Live Project: 8/13/2018 – 8/24/2018</u>

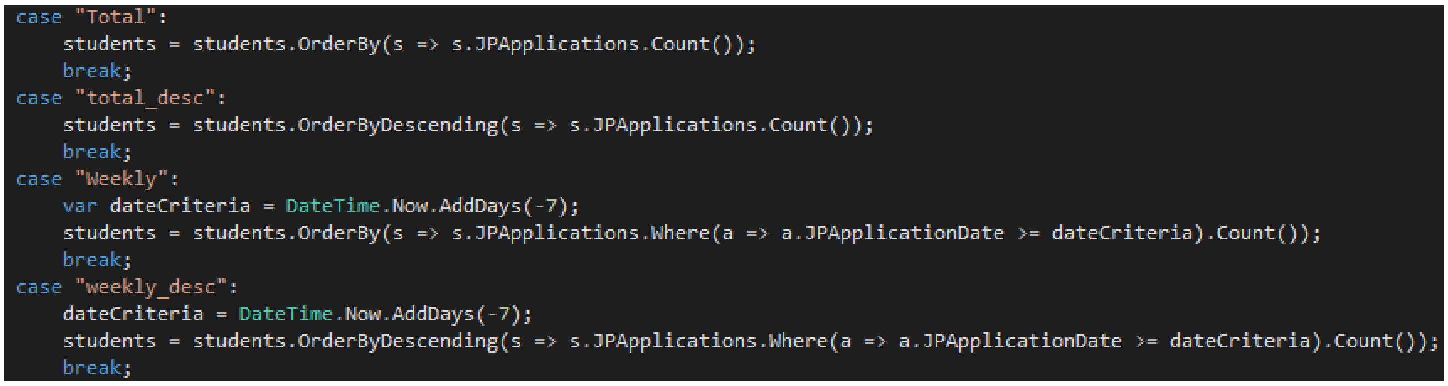
For my first task with the Live Project, I started off simple. The user story requested one of the table headers to be wrapped to two rows so the column could be narrower. Sounds easy enough, right?

What I found is the HTML ActionLink scaffolded by Entity Framework did not allow for certain HTML formatting. My solution was to change it to a URL action link instead, which allowed for a simple HTML line break in the header:



```
<a href="@Url.Action("Index", new { sortOrder = ViewBag.DateSortParm }) ">Days<br/>Active</a>
```

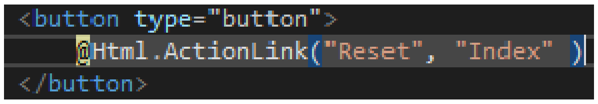
Next, I worked on adding sort functionality to two columns on the page, one that showed the total job applications a student had submitted, and another that showed the total applications submitted within a week of the query date. These two columns did not have a scaffolded HTML ActionLink, only simple column headers with a line break. I had to create links for both headers that corresponded to ascending and descending sort methods on the controller:



```
case "Total":
    students = students.OrderBy(s => s.JPApplications.Count());
    break;
case "total_desc":
    students = students.OrderByDescending(s => s.JPApplications.Count());
    break;
case "Weekly":
    var dateCriteria = DateTime.Now.AddDays(-7);
    students = students.OrderBy(s => s.JPApplications.Where(a => a.JPApplicationDate >= dateCriteria).Count());
    break;
case "weekly_desc":
    dateCriteria = DateTime.Now.AddDays(-7);
    students = students.OrderByDescending(s => s.JPApplications.Where(a => a.JPApplicationDate >= dateCriteria).Count());
    break;
```

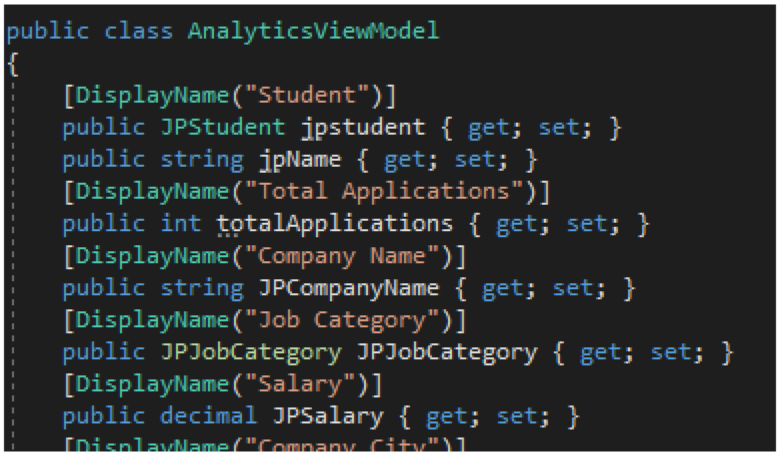
Getting the total applications was easy; it was just a simple Count() per student from the Applications table. Getting the weekly amount was considerably more difficult because I had never used nested lambda expressions before. It was very satisfying when I got it to work properly, however.

The next user story I worked on was to add a reset button to the page, which would clear all filters. I accomplished this with the simple solution of returning the original Index view with a button, requiring no changes to the controller:



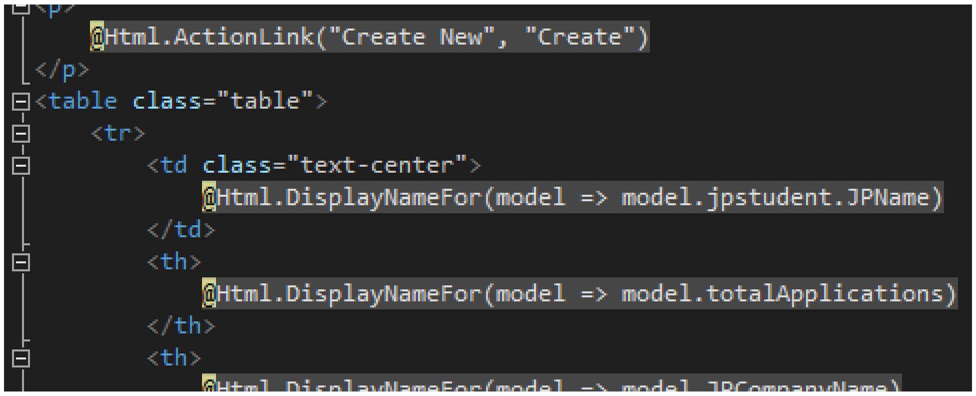
```
<button type="button">
    @Html.ActionLink("Reset", "Index" )
</button>
```

I then began work on the Analytics page, which was several user stories of work. The Job Placement Director would use this page to pull statistics from the database, such as the average number of applications a student submitted before being hired, or the average starting salary of a new hire. First, I created the ViewModel:



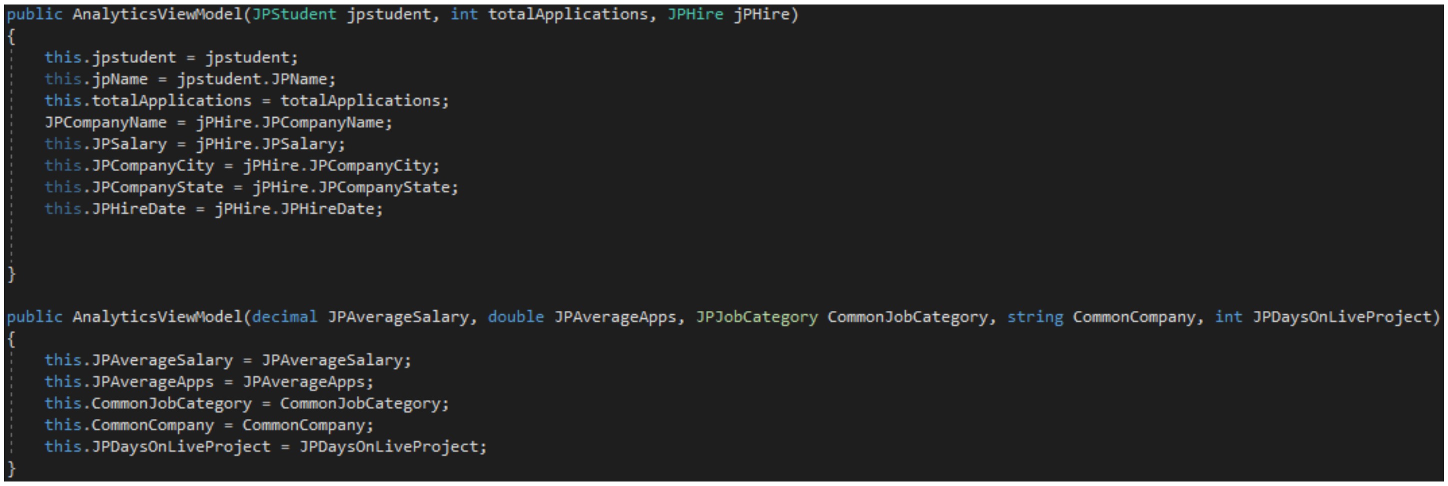
```
public class AnalyticsViewModel
{
    [DisplayName("Student")]
    public JPStudent jpstudent { get; set; }
    public string jpName { get; set; }
    [DisplayName("Total Applications")]
    public int totalApplications { get; set; }
    [DisplayName("Company Name")]
    public string JPCompanyName { get; set; }
    [DisplayName("Job Category")]
    public JPJobCategory JPJobCategory { get; set; }
    [DisplayName("Salary")]
    public decimal JPSalary { get; set; }
    [DisplayName("Company City")]
```

Second, I created the main View. This part was simple because Entity Framework did most of the scaffolding, requiring just a few changes to the View on my part:



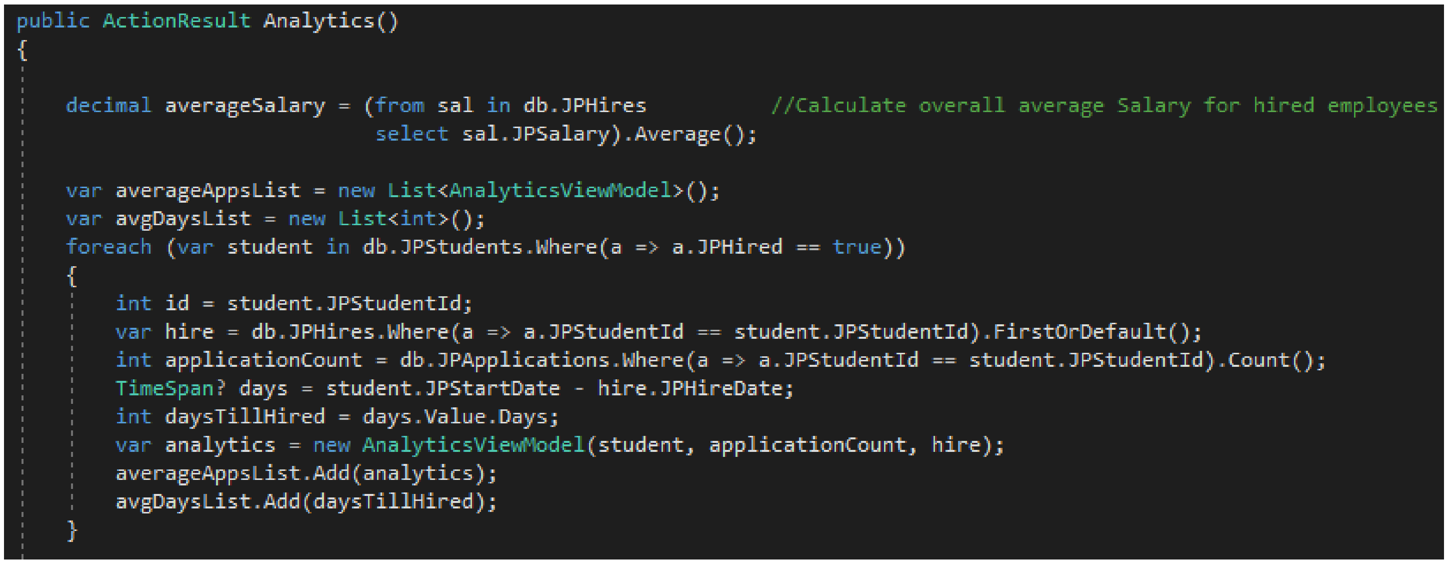
```
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <td class="text-center">
            @Html.DisplayNameFor(model => model.jpstudent.JPName)
        </td>
        <th>
            @Html.DisplayNameFor(model => model.totalApplications)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.JPCompanyName)
```

The next task was to create a constructor for the ViewModel to add more functionality to the controller. We needed an overloaded constructor for this:

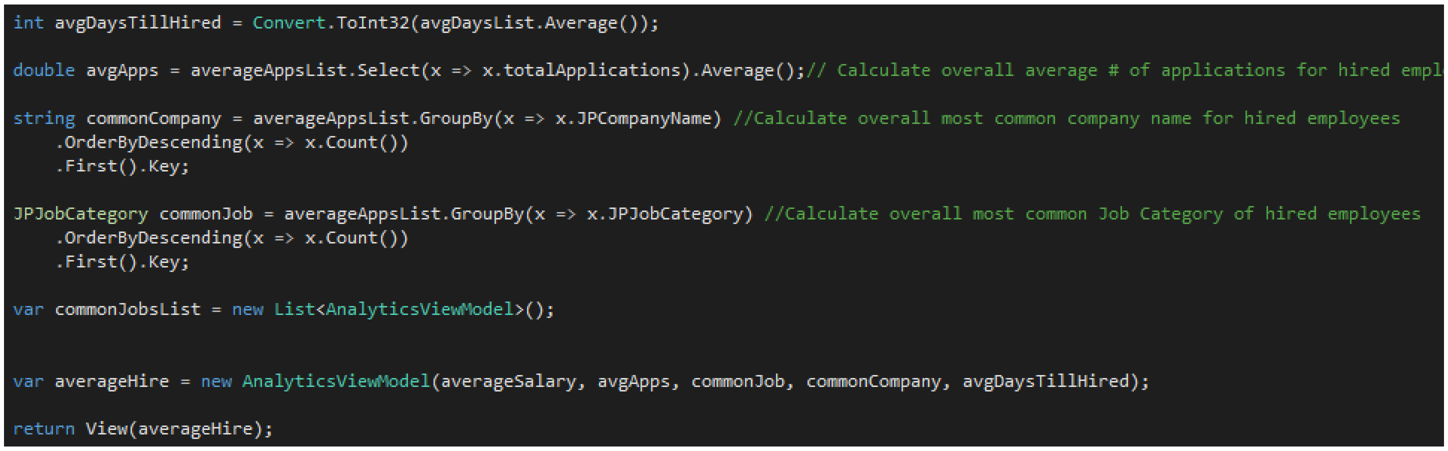


```
public AnalyticsViewModel(JPStudent jpstudent, int totalApplications, JPHire jPHire)
{
    this.jpstudent = jpstudent;
    this.jpName = jpstudent.JPName;
    this.totalApplications = totalApplications;
    JPCompanyName = jPHire.JPCompanyName;
    this.JPSalary = jPHire.JPSalary;
    this.JPCompanyCity = jPHire.JPCompanyCity;
    this.JPCompanyState = jPHire.JPCompanyState;
    this.JPHireDate = jPHire.JPHireDate;


}

public AnalyticsViewModel(decimal JPAverageSalary, double JPAverageApps, JPJobCategory CommonJobCategory, string CommonCompany, int JPDaysOnLiveProject)
{
    this.JPAverageSalary = JPAverageSalary;
    this.JPAverageApps = JPAverageApps;
    this.CommonJobCategory = CommonJobCategory;
    this.CommonCompany = CommonCompany;
    this.JPDaysOnLiveProject = JPDaysOnLiveProject;
}
```

It was exciting to see the constructors in action when writing the controller methods to pull the needed information and post it to the view. This involved some more complicated code, pulling information from multiple database tables using LINQ statements and lambda expressions, adding them to a list of ViewModel objects, then running the calculations off that list.



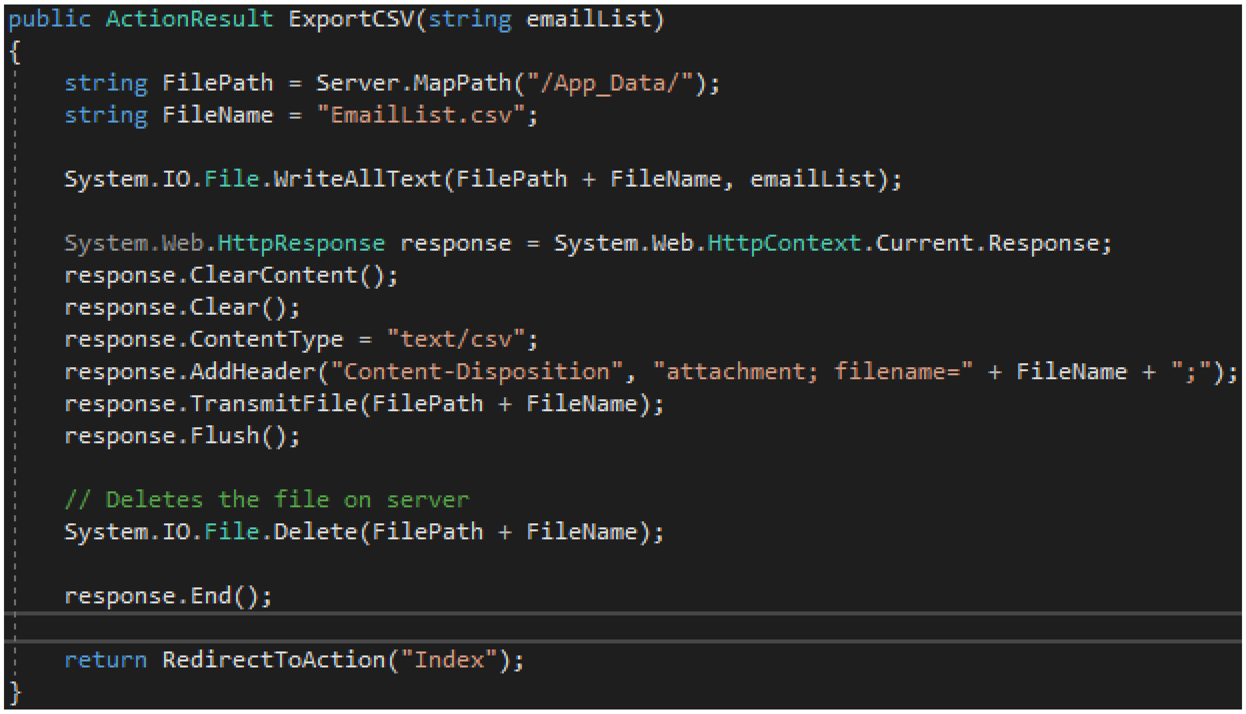
```
public ActionResult Analytics()
{

    decimal averageSalary = (from sal in db.JPHires          //Calculate overall average Salary for hired employees
                             select sal.JPSalary).Average();

    var averageAppsList = new List<AnalyticsViewModel>();
    var avgDaysList = new List<int>();
    foreach (var student in db.JPStudents.Where(a => a.JPHired == true))
    {
        int id = student.JPStudentId;
        var hire = db.JPHires.Where(a => a.JPStudentId == student.JPStudentId).FirstOrDefault();
        int applicationCount = db.JPApplications.Where(a => a.JPStudentId == student.JPStudentId).Count();
        TimeSpan? days = student.JPStartDate - hire.JPHireDate;
        int daysTillHired = days.Value.Days;
        var analytics = new AnalyticsViewModel(student, applicationCount, hire);
        averageAppsList.Add(analytics);
        avgDaysList.Add(daysTillHired);
    }
```



```
int avgDaysTillHired = Convert.ToInt32(avgDaysList.Average());

double avgApps = averageAppsList.Select(x => x.totalApplications).Average();// Calculate overall average # of applications for hired empl

string commonCompany = averageAppsList.GroupBy(x => x.JPCompanyName) //Calculate overall most common company name for hired employees
    .OrderByDescending(x => x.Count())
    .First().Key;

JPJobCategory commonJob = averageAppsList.GroupBy(x => x.JPJobCategory) //Calculate overall most common Job Category of hired employees
    .OrderByDescending(x => x.Count())
    .First().Key;

var commonJobsList = new List<AnalyticsViewModel>();


var averageHire = new AnalyticsViewModel(averageSalary, avgApps, commonJob, commonCompany, avgDaysTillHired);

return View(averageHire);
```

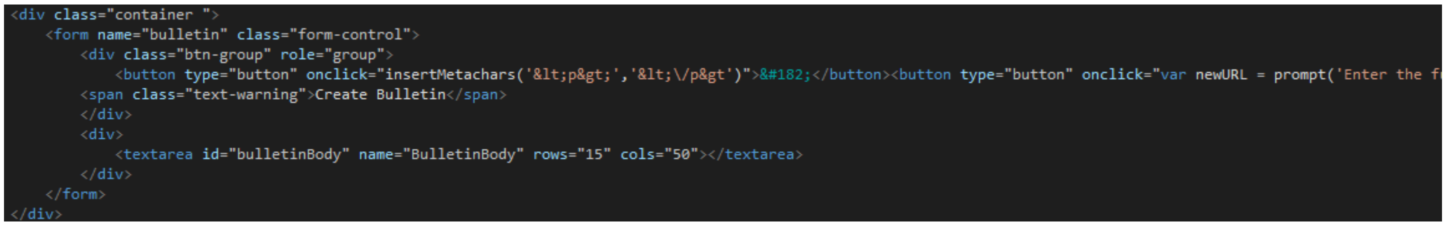
Getting all this data to calculate and post to the view correctly was a proud moment.

My next task was to return to the Student Rundown view to create a button that would export a list of email addresses on the screen as a simple CSV file. Getting a list of student emails to a CSV file was not a problem from a coding perspective. However, getting a list of student emails based on the filtered data on the View was challenging. My working solution was to pass a concatenated string of email addresses, separated by commas, to the controller to export:



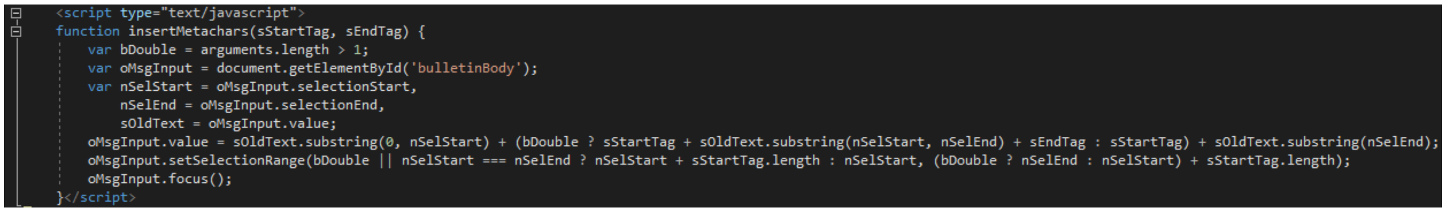
```
public ActionResult ExportCSV(string emailList)
{
    string FilePath = Server.MapPath("/App_Data/");
    string FileName = "EmailList.csv";

    System.IO.File.WriteAllText(FilePath + FileName, emailList);

    System.Web.HttpResponse response = System.Web.HttpContext.Current.Response;
    response.ClearContent();
    response.Clear();
    response.ContentType = "text/csv";
    response.AddHeader("Content-Disposition", "attachment; filename=" + FileName + ";");
    response.TransmitFile(FilePath + FileName);
    response.Flush();

    // Deletes the file on server
    System.IO.File.Delete(FilePath + FileName);

    response.End();

    return RedirectToAction("Index");
}
```

For my next user story, I worked on the Bulletins pages. With these pages, the Job Placement Director would be able to create, edit, and delete posts. Up until this point, virtually all the code we had worked on was in C# and HTML. For this user story, I was tasked with creating buttons on the Create Bulletins view page that would add JavaScript functionality to the page by dynamically adding HTML tags to the text area. It was good to know that I hadn't forgotten my JavaScript knowledge.



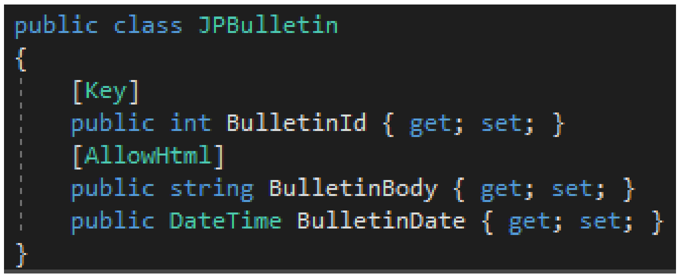
```
<div class="container ">
    <form name="bulletin" class="form-control">
        <div class="btn-group" role="group">
            <button type="button" onclick="insertMetachars('&lt;p&gt;','&lt;\/p&gt')">&#182;</button><button type="button" onclick="var newURL = prompt('Enter the f
        <span class="text-warning">Create Bulletin</span>
        </div>
        <div>
            <textarea id="bulletinBody" name="BulletinBody" rows="15" cols="50"></textarea>
        </div>
    </form>
</div>
```

I added buttons that would add a new paragraph or hyperlink to the post that was being created. When clicked, the buttons would run a script that would add the paragraph and anchor tags to the selected text in the textbox:



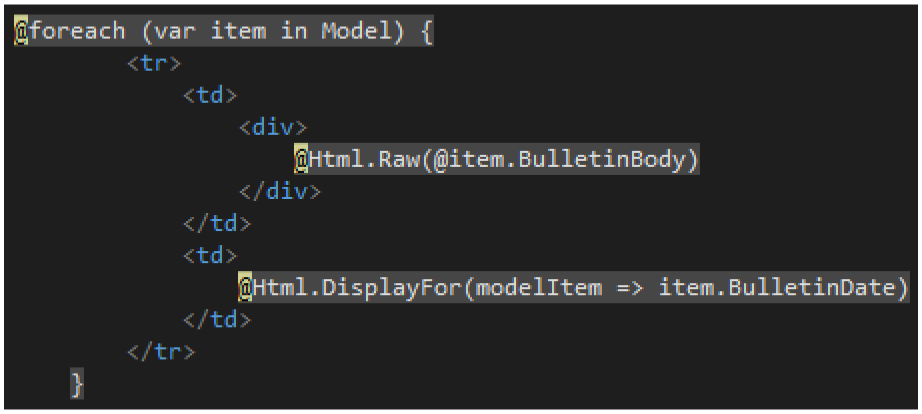
```
<script type="text/javascript">
function insertMetachars(sStartTag, sEndTag) {
    var bDouble = arguments.length > 1;
    var oMsgInput = document.getElementById('bulletinBody');
    var nSelStart = oMsgInput.selectionStart,
        nSelEnd = oMsgInput.selectionEnd,
        sOldText = oMsgInput.value;
    oMsgInput.value = sOldText.substring(0, nSelStart) + (bDouble ? sStartTag + sOldText.substring(nSelStart, nSelEnd) + sEndTag : sStartTag) + sOldText.substring(nSelEnd);
    oMsgInput.setSelectionRange(bDouble || nSelStart === nSelEnd ? nSelStart + sStartTag.length : nSelStart, (bDouble ? nSelEnd : nSelStart) + sStartTag.length);
    oMsgInput.focus();
}</script>
```

The challenge here in getting the script to work was to get the required variables to be recognized as strings. Before I got it to work, I made the mistake of using the “getElementsByClass” method, which returns a list of HTML elements instead of a single element. Once I changed it to “getElementById” and assigned an ID to the HTML element, it worked perfectly – that is, the script worked perfectly. Our team realized the “Create” button stopped working after my update, throwing an error every time the buttons were used. After some quick research, we found the program would not pass HTML tags to the controller for security purposes. Since only the Job Placement Director would have access to the page, I used the simplest solution: adding the “AllowHtml” attribute to the Bulletin Body property:



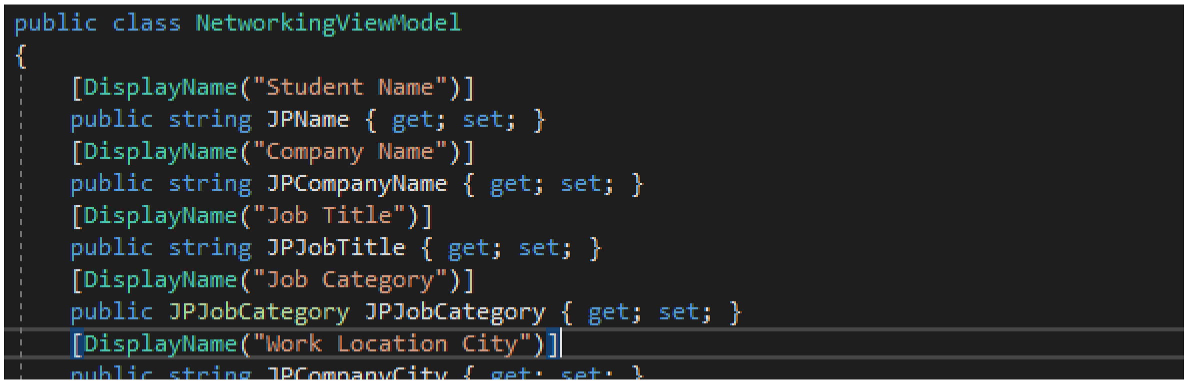
```
public class JPBulletin
{
    [Key]
    public int BulletinId { get; set; }
    [AllowHtml]
    public string BulletinBody { get; set; }
    public DateTime BulletinDate { get; set; }
}
```

This allowed the program to pass HTML tags to the Bulletins page, but the tags were not being interpreted as HTML by the browser, instead posting the HTML code to the bulletin itself. This required a simple change to the Razor syntax on the receiving view by changing it to @Html.Raw:



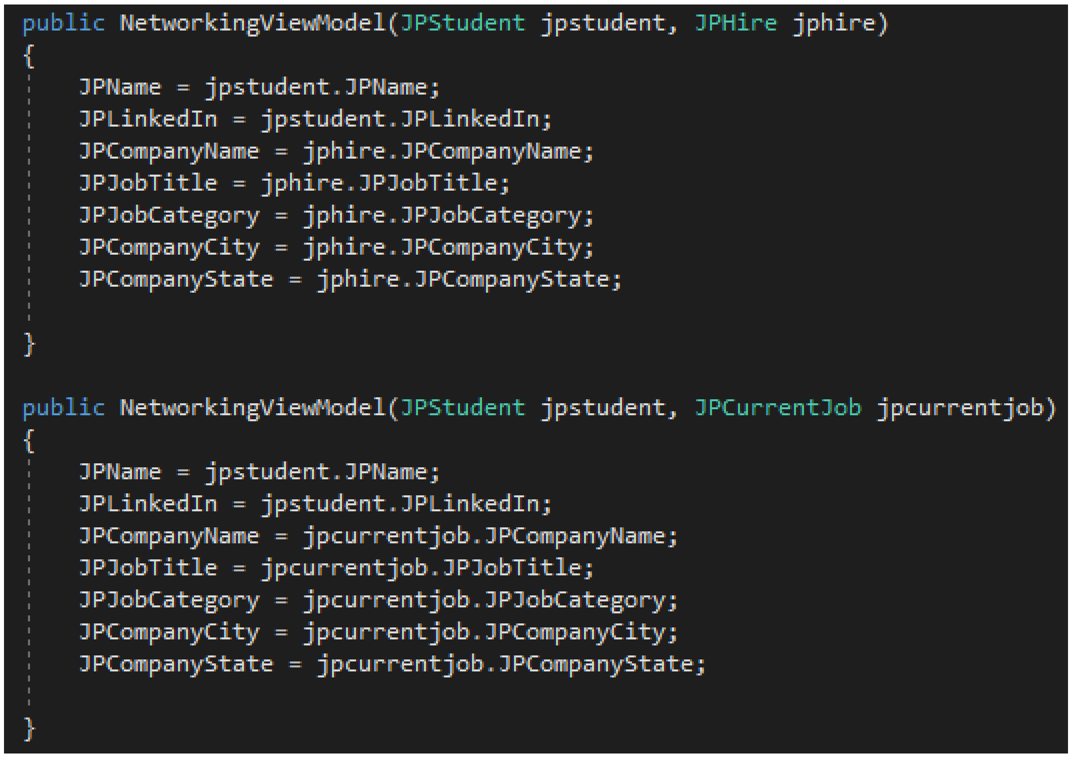
```
@foreach (var item in Model) {
        <tr>
            <td>
                <div>
                    @Html.Raw(@item.BulletinBody)
                </div>
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.BulletinDate)
            </td>
        </tr>
    }
```

My next several user stories involved the Networking pages. I created the ViewModel:



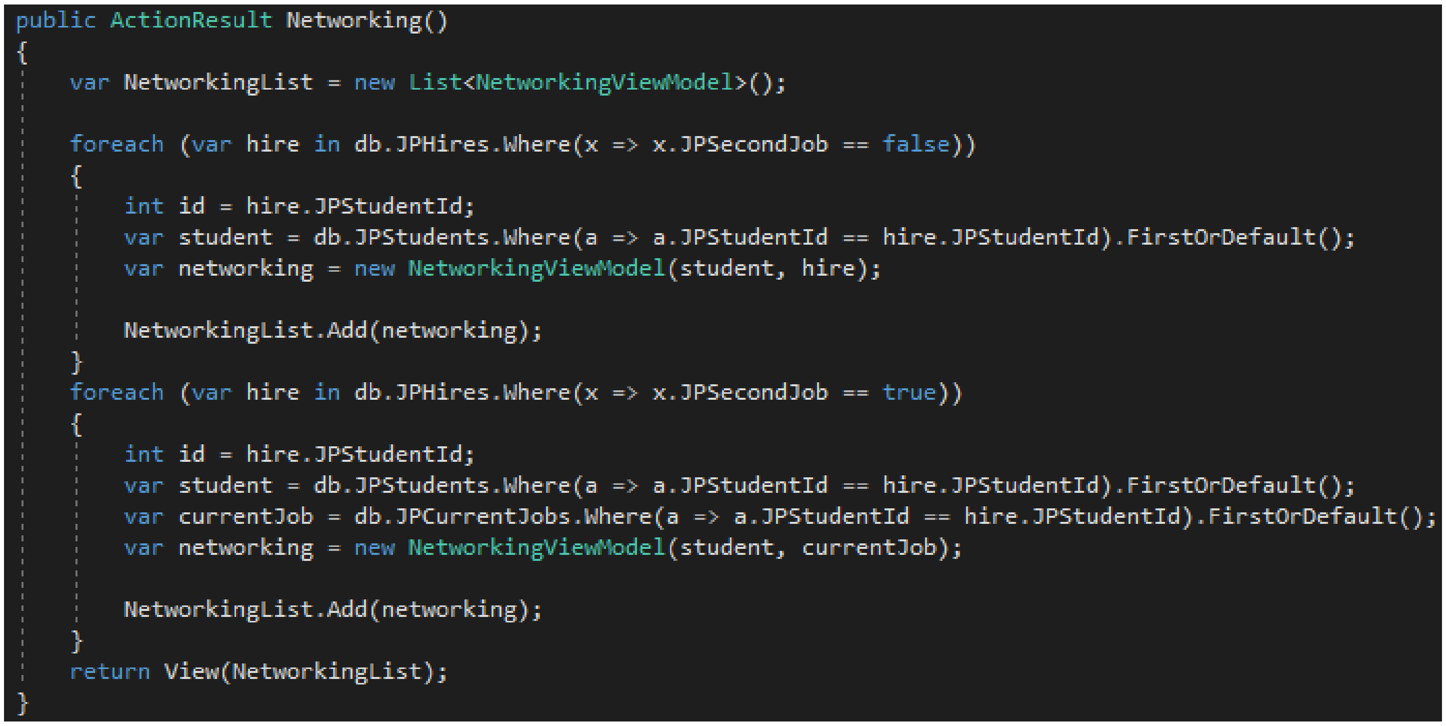
```
public class NetworkingViewModel
{
    [DisplayName("Student Name")]
    public string JPName { get; set; }
    [DisplayName("Company Name")]
    public string JPCompanyName { get; set; }
    [DisplayName("Job Title")]
    public string JPJobTitle { get; set; }
    [DisplayName("Job Category")]
    public JPJobCategory JPJobCategory { get; set; }
    [DisplayName("Work Location City")]
    public string JPCompanyCity { get; set; }
```

After scaffolding the Controller and View, I was tasked with creating a method that would pass a valid object to the view. As with many of the other pages on the site, this involved pulling data from multiple tables in the database. I accomplished this by creating an overload constructor:



```
public NetworkingViewModel(JPStudent jpstudent, JPHire jphire)
{
    JPName = jpstudent.JPName;
    JPLinkedIn = jpstudent.JPLinkedIn;
    JPCompanyName = jphire.JPCompanyName;
    JPJobTitle = jphire.JPJobTitle;
    JPJobCategory = jphire.JPJobCategory;
    JPCompanyCity = jphire.JPCompanyCity;
    JPCompanyState = jphire.JPCompanyState;

}

public NetworkingViewModel(JPStudent jpstudent, JPCurrentJob jpcurrentjob)
{
    JPName = jpstudent.JPName;
    JPLinkedIn = jpstudent.JPLinkedIn;
    JPCompanyName = jpcurrentjob.JPCompanyName;
    JPJobTitle = jpcurrentjob.JPJobTitle;
    JPJobCategory = jpcurrentjob.JPJobCategory;
    JPCompanyCity = jpcurrentjob.JPCompanyCity;
    JPCompanyState = jpcurrentjob.JPCompanyState;

}
```

The first would pull the required data from the Students and Hires tables, and the second would pull the required data from the Students and Current Jobs tables, which was used if a hire had a second job. The controller would post these both to the view as a single list. I wrote the following:



```
public ActionResult Networking()
{
    var NetworkingList = new List<NetworkingViewModel>();

    foreach (var hire in db.JPHires.Where(x => x.JPSecondJob == false))
    {
        int id = hire.JPStudentId;
        var student = db.JPStudents.Where(a => a.JPStudentId == hire.JPStudentId).FirstOrDefault();
        var networking = new NetworkingViewModel(student, hire);

        NetworkingList.Add(networking);
    }
    foreach (var hire in db.JPHires.Where(x => x.JPSecondJob == true))
    {
        int id = hire.JPStudentId;
        var student = db.JPStudents.Where(a => a.JPStudentId == hire.JPStudentId).FirstOrDefault();
        var currentJob = db.JPCurrentJobs.Where(a => a.JPStudentId == hire.JPStudentId).FirstOrDefault();
        var networking = new NetworkingViewModel(student, currentJob);

        NetworkingList.Add(networking);
    }
    return View(NetworkingList);
}
```

At this point, I was confident in the code I had written, but was not able to test it because we didn't have anything in the Current Jobs table of our test database. I liked knowing exactly what the code would do if it were run. Once we got the test data populated, and I was finally able to test my code, it was satisfying to see it go off without a hitch.