INT422 Assignment 4

Work with associated data, display-only.

Read/skim all of this document before you begin work.

Due date

Section A: Thursday, June 9, 2016, at 11:59pm ET

Grade value: 4% of your final course grade

If you wish to submit the lab before the due date and time, you can do that.

Objective(s)

Work with associated data, display only. Your app will enable users to view a list of invoices, and detail for an invoice. The detail will be extensive, and will include data from many associated entities.

Introduction to the problem to be solved

We need an “invoice viewer” web app. The data shown by the invoice viewer must be rich and extensive.

Specifications overview and work plan

Here’s a brief list of specifications that you must implement:

- Follows best practices
- Implements the recommended system design guidance
- Customized appearance, with an added menu item
- Displays invoice and associated data

Here is a brief work plan sequence:

1. Create the project, based on the project template
2. Customize the app’s appearance

3. Create view models and mappers that cover the initial use cases
4. Add methods to the Manager class that handle the use cases
5. Add controller(s), with code to work with the manager object
6. For the invoice entity, implement the “get all” use case; including controller code, and view
7. Implement the “get one” use case; including controller code, and view
8. Incrementally add associated data

During the class/session, your professor will help you *get started* and *make progress* on this assignment.

Every week, in the computer-lab class/session, your teacher will record a grade when you complete a specific small portion of the assignment. We call this “*in-class grading*“.

The *in-class grading* will be announced in-class by your professor.

Getting started

Create a new web app, named Assignment4.

It MUST use the new “Web app project v2” project template. Download this new project template from the course website, and install it into your Visual Studio configuration.

Warning: Your teachers believe that the best way to work through this Assignment 4 is to do it *incrementally*. Get one thing working, before moving on to the next. Test each part.

Customize the app’s appearance

You will customize the appearance all of your web apps and assignments. Never submit an assignment that has the generic auto-generated text content. Make the time to customize the web app’s appearance.

For this assignment, you can defer this customization work until later. Come back to it at any time, and complete it before you submit your work.

Follow the guidance from Assignment 1 to customize the app’s appearance.

Create view models and mappers that cover the initial use cases

We will be working with the invoice entity, AND some of its associated entities.

Tip:

> Study the DesignModelClasses.cd class diagram that's in the Models folder.
>
> It will help you visualize where the Invoice entity is located in the design model.

As noted above, these use cases need view models:

- Invoice – "get all"
- Invoice – "get one"

Go ahead and write those view model classes. Do not worry about associated entities yet. Focus only on the "get all" and "get one" for the invoice entity.

Remember to add the [Key] data annotation to all/most of your view model classes.

Mappers

Define the maps that these use cases will need. At this point in time, you should have enough experience to know which maps are required. Ask if you need help.

Add methods to the Manager class that handle the use cases

Again, focus only on "get all" and "get one", for the invoice entity. Do not worry about associated entities yet.

In the Manager class, add the methods that support these use cases.

For "get all", you should probably use a LINQ query expression to sort the results in a logical way.

Add controllers, with code to work with the manager object

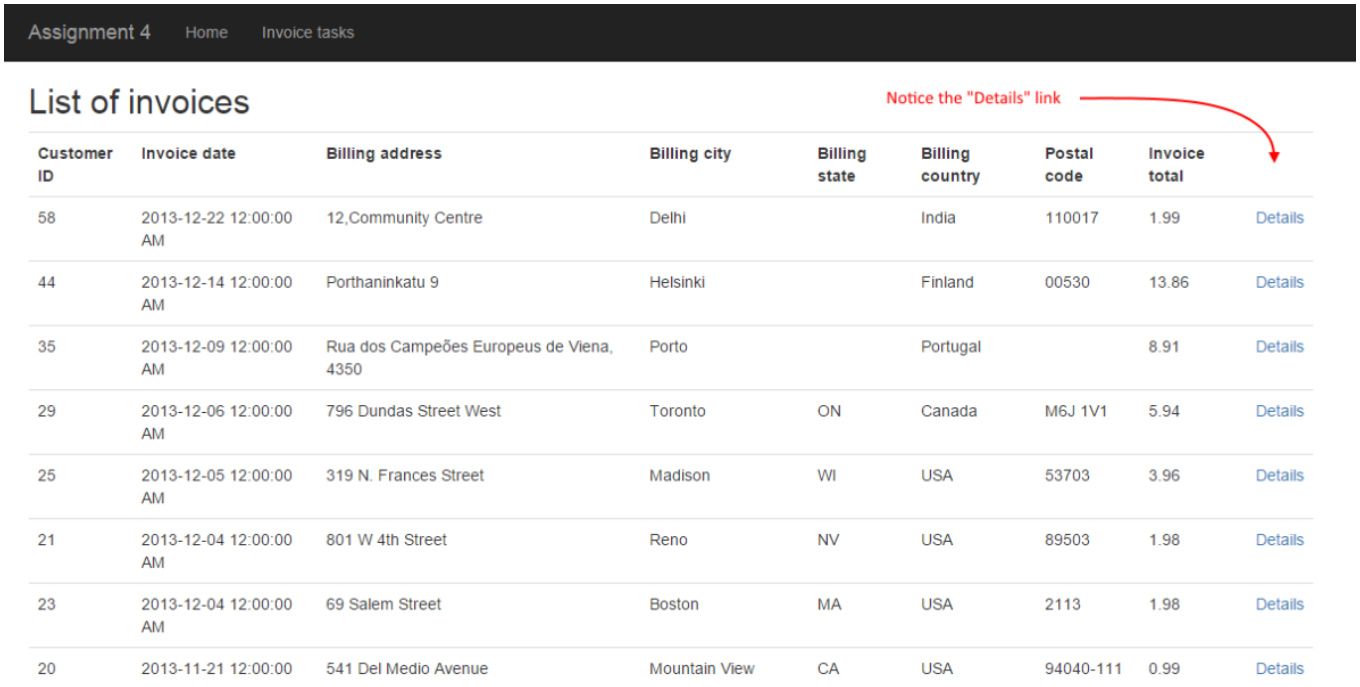
Again, focus only on "get all" and "get one", for the invoice entity. Do not worry about associated entities yet.

Create a controller for the invoice entity.

For the invoice entity, implement the “get all” use case; including controller code, and view

The “get all” use case will show the default view when working with invoice objects. Each invoice will have a “Details” link at the right side.

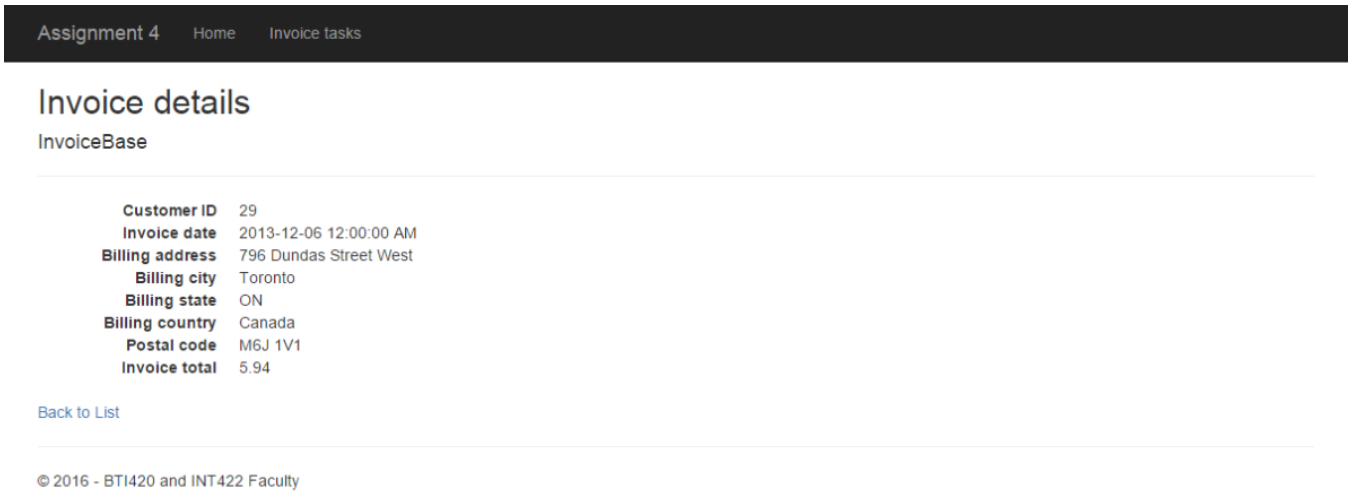
Here’s an example screen capture.



Implement the “get one” use case; including controller code, and view

The “get one” use case will show the default ugly view for an invoice object.

Here’s an example screen capture.



Incrementally add associated data

At this point, your app will reliably display a list of invoices, and individual invoices.

Now, it's time to incrementally add associated data. We will add some data from these entities:

- Customer
- Employee (via Customer)
- InvoiceLine
- Track (via InvoiceLine)
- Album (via Track and InvoiceLine)
- MediaType (via Track and InvoiceLine)
- Artist (via Album and Track and InvoiceLine)

We will use the same sequence from the "Specifications overview and work plan" section above.

Add data from Customer (and Employee)

Do we need "customer" view model classes? Interestingly, we do not, if we used composed property names and AutoMapper's "flattening" feature.

So, let's proceed with that plan. It will less our workload.

Create an InvoiceWithDetail view model class. It will inherit from InvoiceBase.

Add some composed property names from the Customer and Employee design model classes.

In a class, a composed property name is a string concatenation of the *property* names, from the navigation property in the current class, to the property in the destination class. On the way to the destination, you can pass through other navigation properties.

For example, the Invoice class has a "Customer" navigation property. If we want the Customer class "FirstName" property, the composed property name will be "CustomerFirstName".

In addition, the Customer class has an "Employee" navigation property. If we want the Employee class "FirstName" property, the composed property name will be "CustomerEmployeeFirstName".

For now, we need:

- customer first and last names, and city, and state
- employee (i.e. the customer's sales rep) first and last name

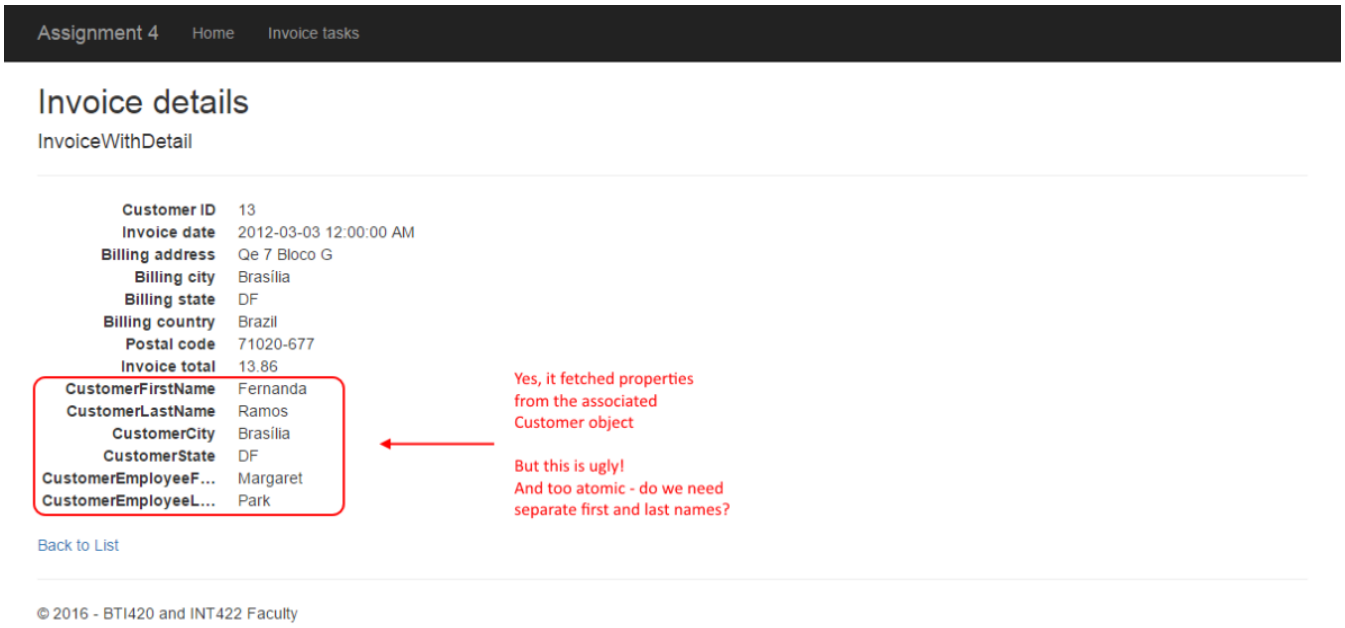
Add a mapper, from Invoice, to InvoiceWithDetail.

In the Manager class, add another "get one" invoice method. Its purpose will be to fetch an invoice, with some associated data. Its suggested name is "InvoiceGetByIdWithDetail()". It must use the Include() method, for the Customer and Employee properties.

How can you get employee information?

> The problem is that Invoice does NOT have a direct association with Employee. Instead, the path is Invoice > Customer > Employee.
>
> Can we easily get employee information? Yes. You may recall reading the MSDN documentation for the Include() method. The "path" parameter is a "dot-separated list of related objects to return in the query results."
>
> Ah. Something like "Customer.Employee".

At this point in time, you have had something that looks like this.



Can we make it look better?

It's ugly. Can we make it look better? Yes. Hand-edit the view:

- Format the numbers and dates so they look nicer
- Gather the invoice-specific info together (and display the invoice identifier)
- Gather the customer info together

<u>Format the numbers and dates</u>

At the top of every view, there is a Razor code expression block (which starts with @{…).

We suggest that you add code to this block, to declare and prepare strings that can be used in the view. For example, we can format dates and numbers. And we can concatenate strings in a more convenient manner.

Note: There is a DisplayFormat data annotation.
That can be quite useful for date and number formatting too.

To get you started, let's change the ugly default date format (for the invoice date), to something that's nicer to read and understand.

Here's the edited code block:

```
@{
    ViewBag.Title = "Invoice details";

    // Prepare some strings, to be used below
    var invoiceDate = Model.InvoiceDate.ToLongDateString();

    // Add more here
}
```

Then later, when you need to use the new pretty "invoiceDate":

```
<dd>
    @invoiceDate
    @*@Html.DisplayFor(model => model.InvoiceDate)*@
</dd>
```

Gather the invoice-specific info together (and display the invoice identifier)

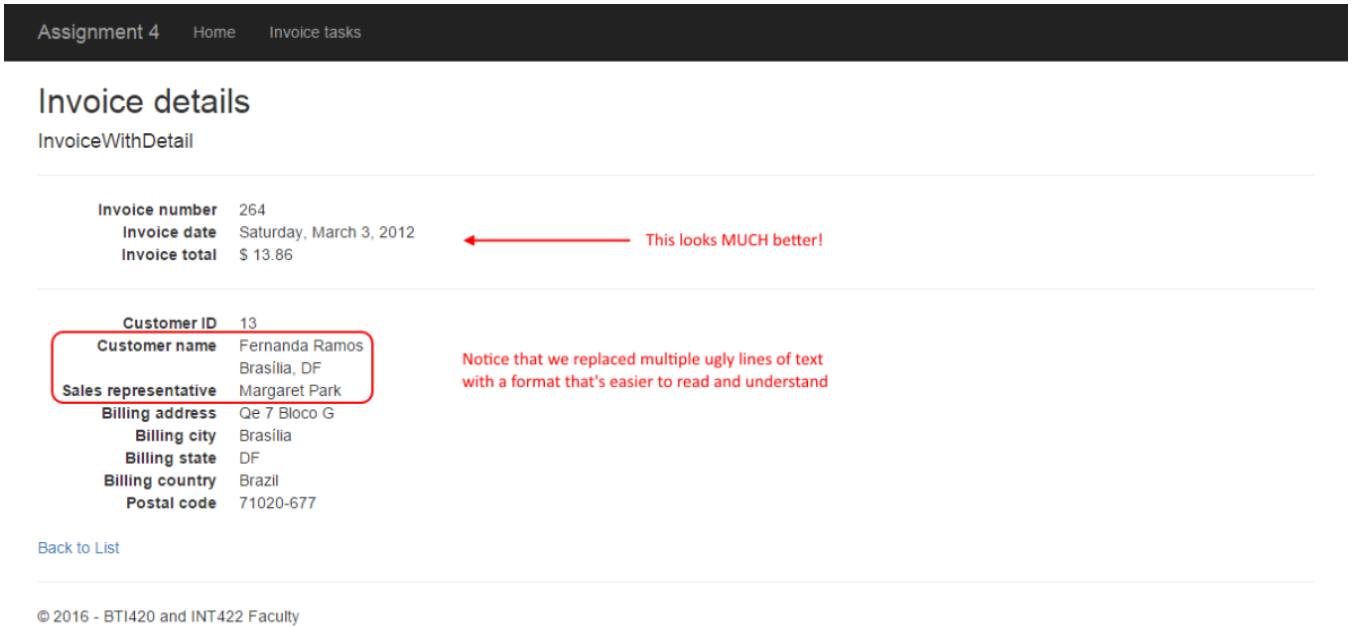
We suggest that you create a new <dl> element, to hold the invoice-specific info.

Remember – a view *source code file* can hold HTML and code expressions.

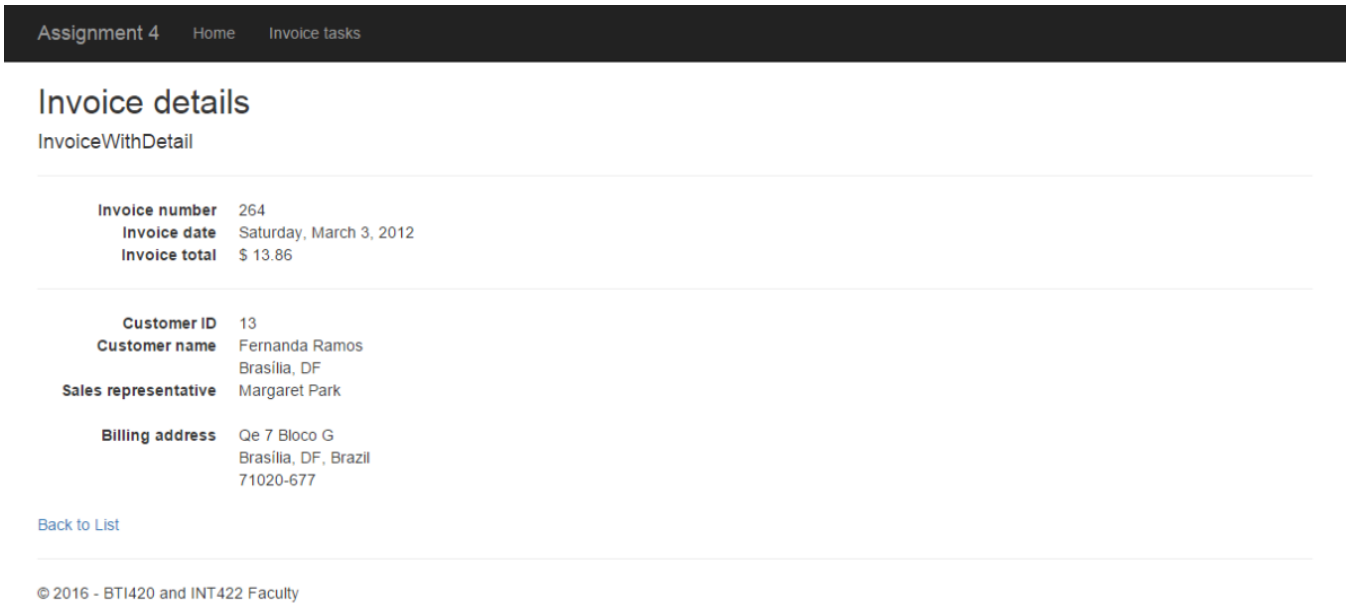
Gather the customer info together

Maybe the first step is to get rid of those atomic customer-related elements. Create new concatenated strings (in the code block at the top of the view).

At this point in time, you have have something that looks like this. Looks pretty good – almost perfect (for now).



Finish off the work by following the same process to collapse and format the billing address info. You may end up with something like this.



Add data from InvoiceLine

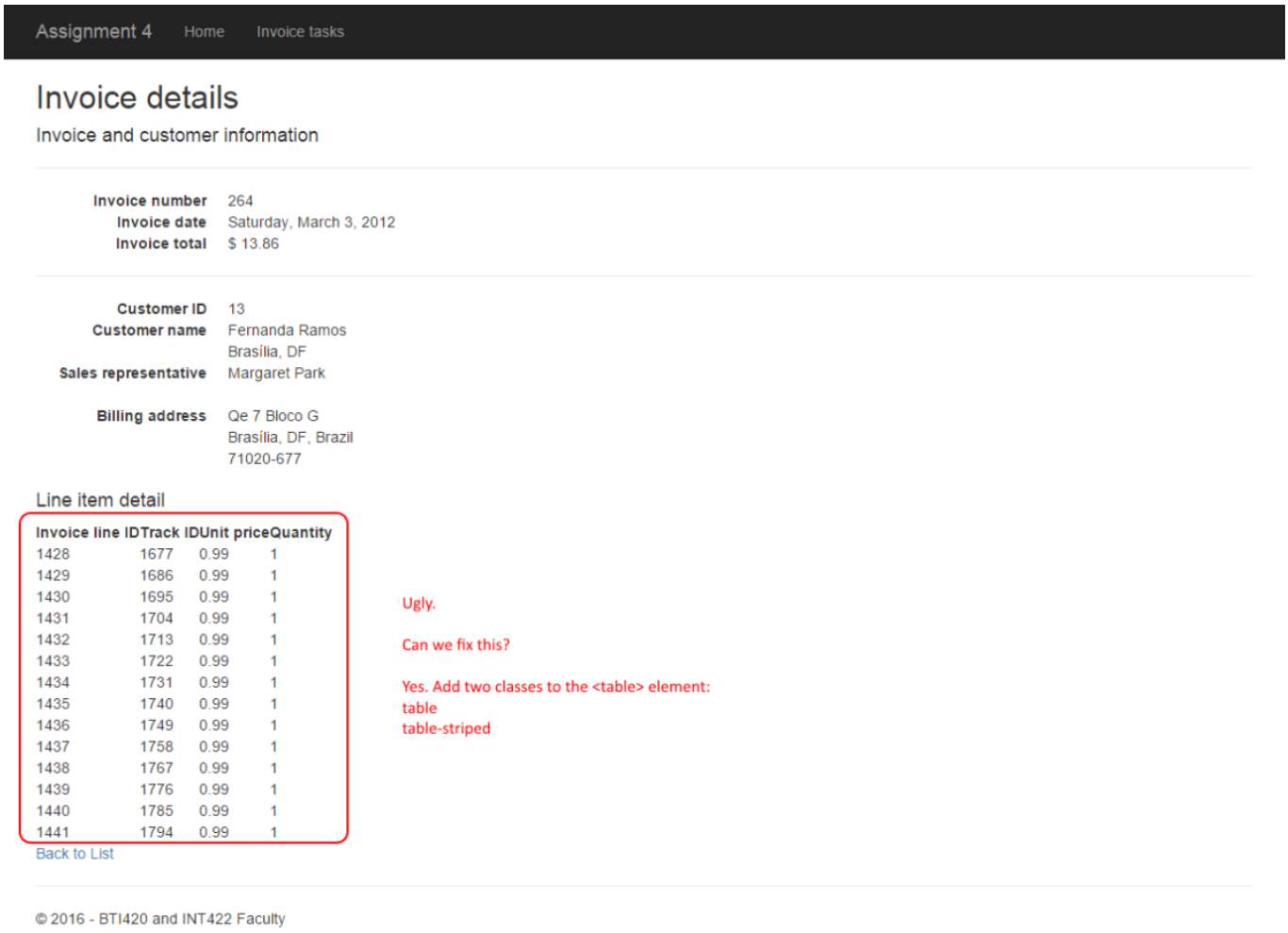
Now, we will add invoice line-item detail.

First, we need a “base” view model class for InvoiceLine. And a mapper.

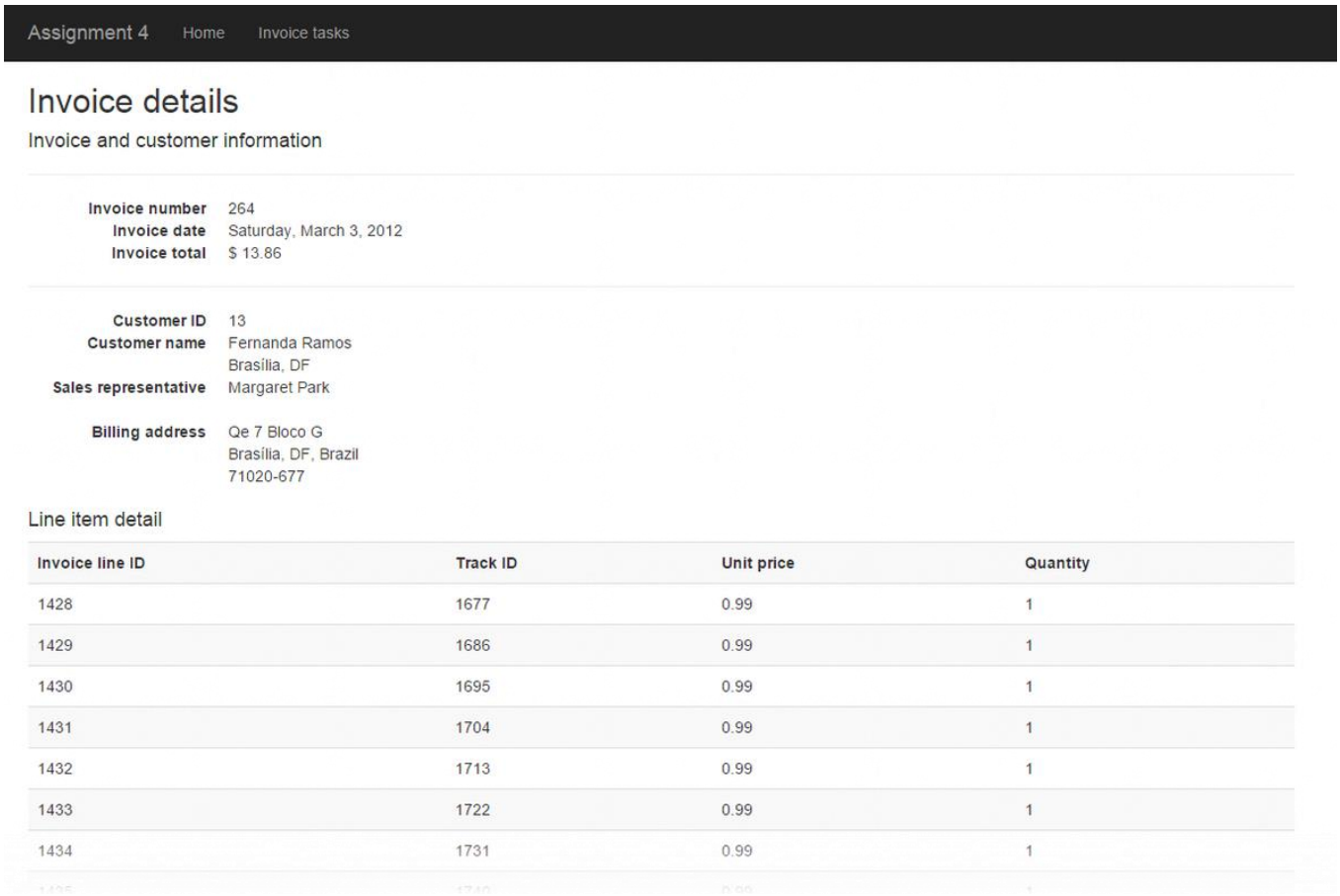
Next, modify the invoice-with-details view model class, by adding a collection navigation property for the InvoiceLineBase items. Make sure the name of this navigation property matches the design model class.

Now, modify the manager invoice-get-by-id-with-detail method. Include the InvoiceLines property in the statement. The controller method is good as is, with no changes.

Finally, in the view source code, add HTML and code expressions to build a table, located below the existing information. It will probably look like the following.



Not bad, but the table is ugly. Follow the guidance in the image above to fix it. The two classes (table and table-striped) are defined by the Bootstrap framework (which will be introduced soon in this course). Then, it will probably look something like the following.



Line item calculation

Add another column, to hold the line item total.

Add data from Track (via InvoiceLine)

While the invoice line items are technically correct, they're not very useful to a browser user. We must add information about the track that's readable and understandable for the browser user. For example, the track name, and other detail.

A strategy similar to what we did for Customer and Employee (above) is recommended. We'll take advantage of the AutoMapper "flattening" feature.

Create an InvoiceLineWithDetail view model class. It will inherit from InvoiceLineBase.

Add some composed property names from the Track design model class. We want the track name, for sure. Others? We could optionally add the composer.

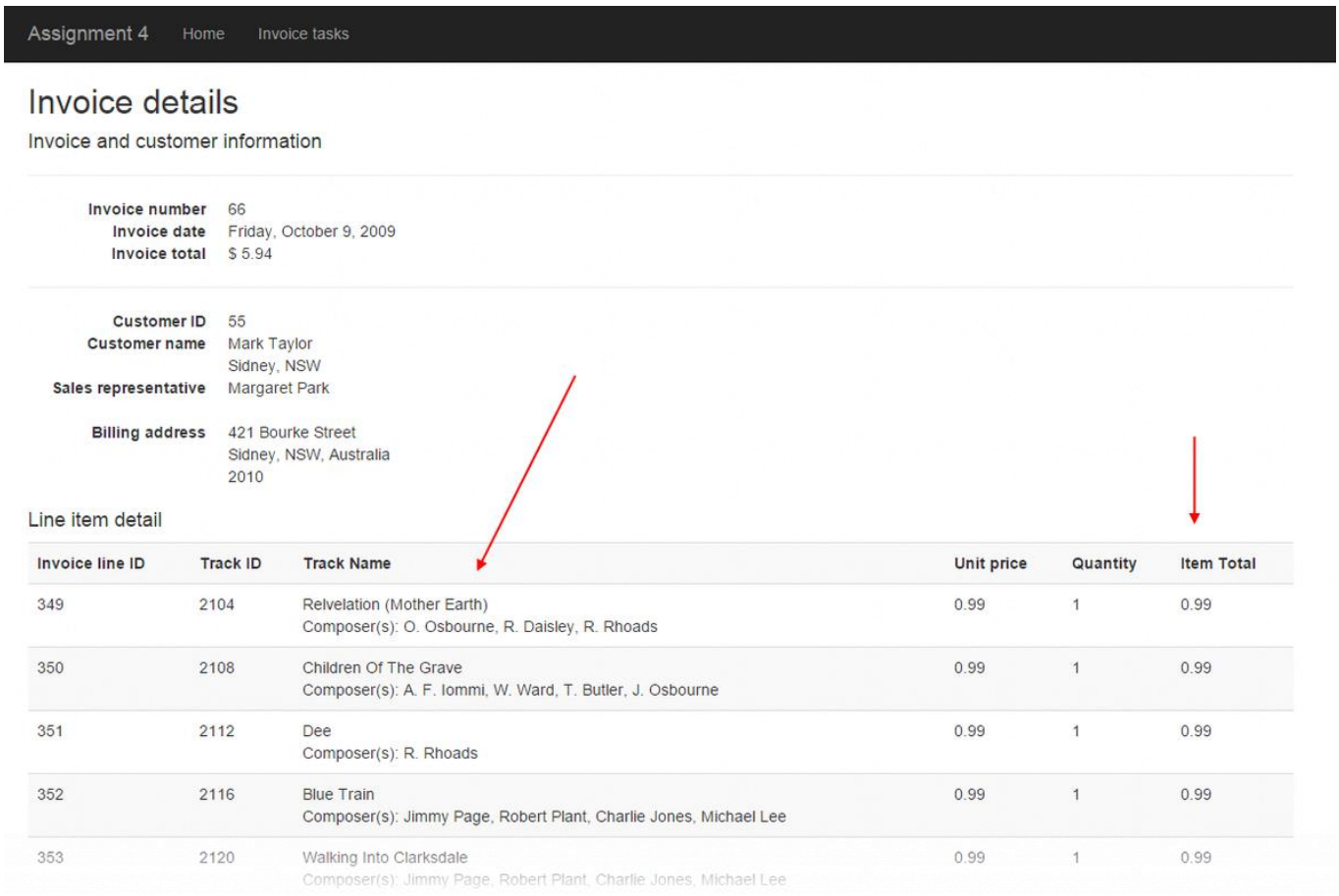
Add a mapper, from InvoiceLine to InvoiceLineWithDetail.

Modify the existing manager method so that it includes the path InvoiceLines > Track.

Next, go back to the *invoice* view model classes. The invoice-with-details view model class has a navigation property that holds the collection of invoice lines. What's its data type? InvoiceLineBase. We must change the data type, to the new view model class that you just created (InvoiceLineWithDetail).

Finally, edit the view source code.

At this point, you will probably have something that looks like the following.



Add data from Album (via Track and InvoiceLine) and data from Artist (via Album and Track and InvoiceLine)

We hope that you now see a pattern for working with associated data. And that adding data from the Album and Artist objects is surprisingly easy.

Track has a to-one association with Album. To work with the Album “Title” property:

1. Add a composed property name to the InvoiceLineWithDetail view model class
2. Modify the existing manager method, to include the *path* to the Album object
3. Edit the view source code

Album has a to-one association with Artist. To work with the Artist “Name” property:

1. Add a composed property name to the InvoiceLineWithDetail view model class
2. Modify the existing manager method, to include the *path* to the Artist object
3. Edit the view source code

Add data from MediaType (via Track and InvoiceLine)

Above, when working with Album and Artist data, we had a straight-line path from the InvoiceLine object:

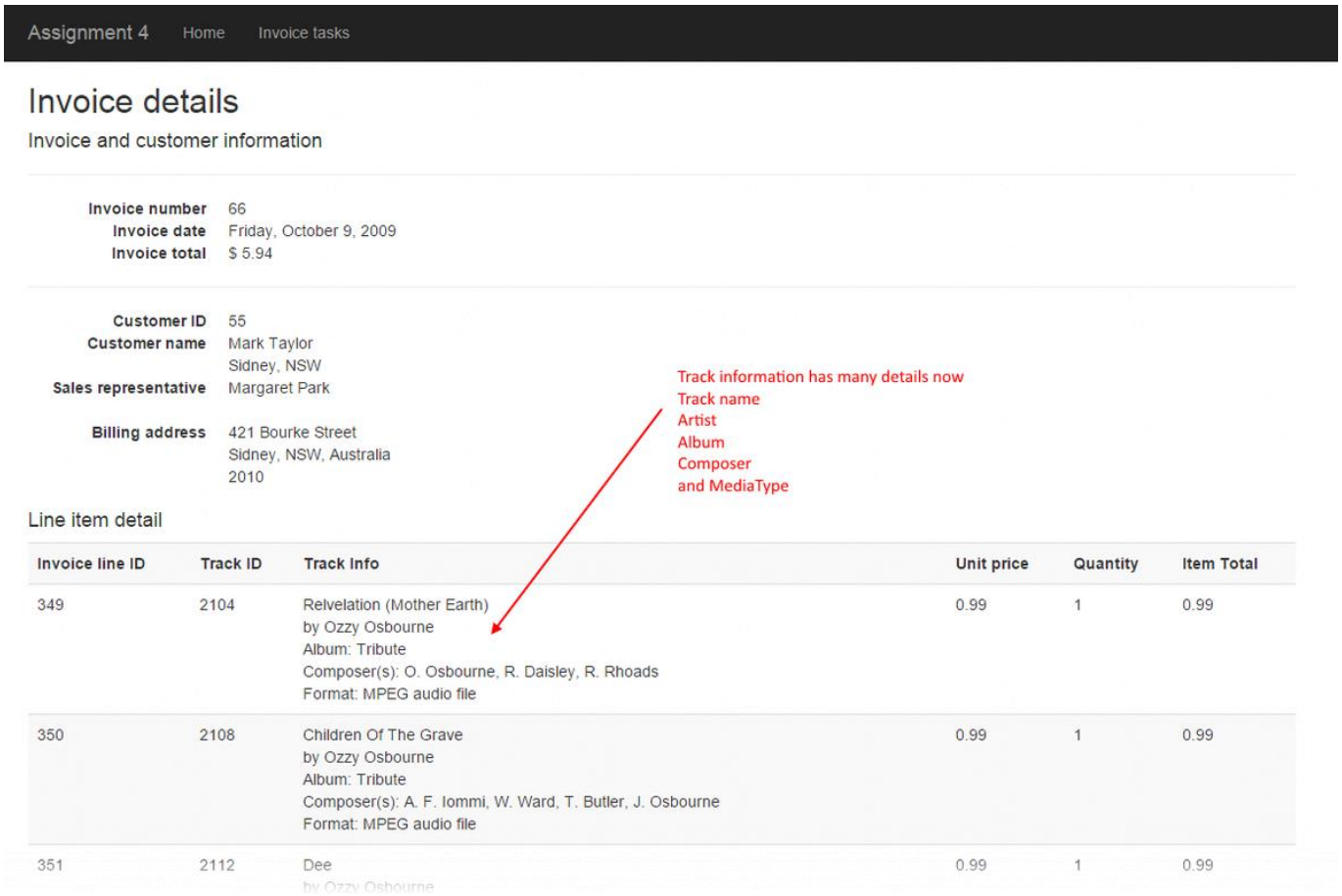
InvoiceLine > Track > Album > Artist

Now, we need a piece of data from the MediaType object. It has a slightly different path:

InvoiceLine > Track > MediaType

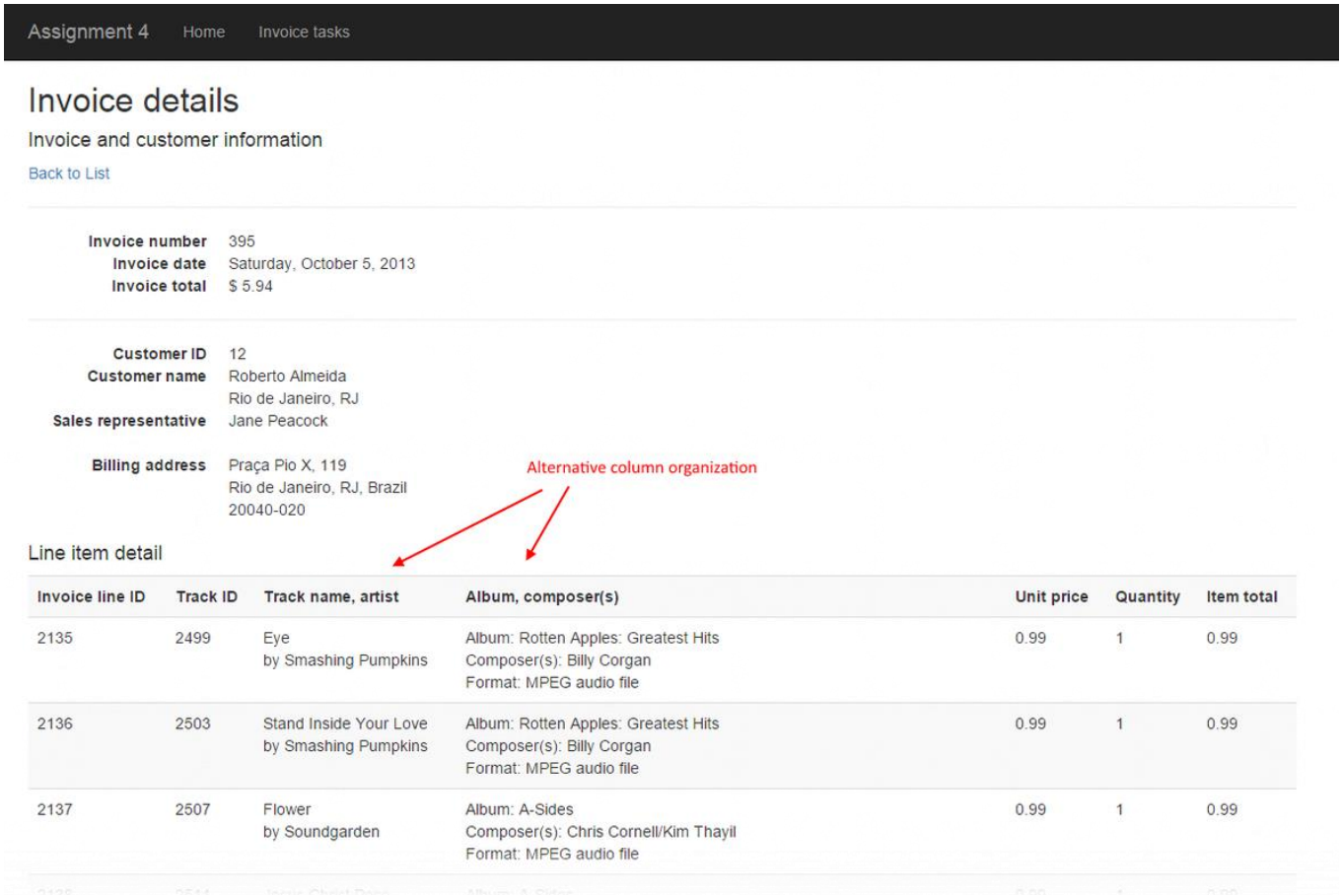
As a result, to add data from MediaType (the “Name” property), we still follow the same strategy as we did with Album and Artist, but we will need a third Include() method to get to the MediaType object.

If you did this correctly, you will probably have something that looks like the following.



Alternative column organization

Just for fun, you can play around with the view source code, and organize the columns in different ways. For example, the following shows two columns to hold track information.



Testing your work

In a browser, test your work, by doing tasks that fulfill the use cases in the specifications.

Reminder about academic honesty

You must comply with the College's academic honesty policy. Although you may interact and collaborate with others, *you must submit your own work*.

Submitting your work

Here's how to submit your work, before the due date and time:

1. Locate the folder that holds your solution files. In Solution Explorer, right-click the "Solution" item, and choose "Open Folder in File Explorer". It has three (or more) items: a Visual Studio Solution file, a folder that has your project's source code, and a "packages" folder. Go UP one level.

2. Make a copy of the folder. This is the version that you will be uploading.

3. Remove the "packages" folder from the copied folder; also, remove the "bin" and "obj" folders.

4. Compress/zip the copied folder. The zip file SHOULD be about 2MB or less in size. If it isn't, you haven't followed the instructions properly.

5. Login to My.Seneca/Blackboard. Open the Web Programming on Windows course area. Click the "Assignments" link on the left-side navigator. Follow the link for this lab. Submit/upload your zip file. The page will accept three submissions, so if you upload, then decide to fix something and upload again, you can do so.