**DPS907 WSA500 Assignment 3**

More work with associated data. And, work with non-text media items.

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

**Due date**

Thursday, October 4, 2018, at 11:00pm ET

Grade value: 10% of your final course grade

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

**Overview and purpose**

We need a web service that will deliver data about the music business. Your app will enable users to request data about invoices, and detail for a specific invoice. The detail will be extensive, and will include data from many associated entities. It will also enable you to add employees, and configure them with photos.

**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

1. Customized appearance on the landing web page
2. Uses Entity Framework and Code First technology
3. Includes a Fiddler log file that shows complete coverage of tests
4. Support HTTP GET, for the *get all* use case, for the Invoice entity
5. Support HTTP GET, for *get all*, for Invoice
6. Support HTTP GET, for *get one*, for Invoice, with info from related objects
7. Support HTTP GET, for *get one*, for Employee, with info from related objects
8. Support HTTP POST, for the *add new* use case, for Employee
9. Implement the command pattern, to enable the configuration of an employee’s manager
10. Accept, upload, and deliver non-text internet media types (employee photos)
11. Implement basic content negotiation

Students in DPS907 must complete these additional specifications (use cases):

* Implement the command pattern, to enable the configuration of an employee’s direct-reports
* Enable the use of an Album cover art image
* Enable the use of an audio Track sample
* Enable the use of a PDF version of an Invoice
* Implement beyond-basic content negotiation

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, it is possible that your professor 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.

**Review the concepts if you need to**

Whether you have been away from ASP.NET MVC web app programming for one or two academic terms, you should review some notes, to ensure that you use best practices and efficient design and coding techniques.

Our course notes have coverage, and more can be found in the professor’s recent [web apps course notes](https://petermcintyre.com/bti420/notes/).

Also, remember and apply the knowledge and techniques that you’ve recently re-learned, including the design and coding of resource model classes, AutoMapper configuration and usage, and working with associated data.

**Getting started**

Create a new web service, named Assignment3. It must use the “Web service project v1” project template.

Build/compile, and run (without debugging), to ensure that the app’s home > index view loads in a browser. Then, customize the index view (with your name etc.) before continuing.

**Doing the work, for text (JSON) interactions**

After configuring and studying the app’s structure, you can circle back and do the programming work.

As a reminder, your professor believes that the best way to work through any assignment is to do it *incrementally*. Get one thing working correctly, before moving on to the next. Test each part.

Therefore, you will probably be cycling through the following programming tasks, *for each use case*.

1. Write the resource model class(es) that support the use case
2. Add one or more AutoMapper mappings for the class(es) that you just added
3. Write the Manager class method that performs the data service operation
4. Create a controller, if necessary
5. Write the controller method that implements the use case
6. Test, using Fiddler, debug and fix if necessary

Incrementally doing these tasks will ensure that you’re making progress. You will also notice that they’re repetitive. That’s good, because the process will become familiar and natural to you.

Here are the use cases, from above, for text (JSON) interactions:

* Support HTTP GET, for *get all* use cases, for Invoice and Employee entities
* Support HTTP GET, for *get one*, for Invoice, with info from related objects
* Support HTTP GET, for *get one*, for Employee, with info from related objects
* Support HTTP POST, for the *add new* use case, for Employee
* Implement the command pattern, to enable the configuration of an employee’s manager

And, for students in DPS907, complete these additional specifications:

* Implement the command pattern, to enable the configuration of an employee’s direct-reports

**Info about the *get all* use cases**

A common practice for *get all* use cases is to return a collection of objects, without associated objects. That’s a good starting point. And, that’s what you can do when returning the Invoice collection, and the Employee collection.

However, when appropriate, some associated data can be returned. That data can be individual/selected properties (mapped using the composed pattern/feature of AutoMapper, also known as flattening), or less frequently, a nested object or collection. Also, you could create new calculated/computed properties, if that makes sense.

For example, a collection of invoice data is pretty boring. It could be more meaningful to the requestor if, for example, the customer name was included, instead of just the customer identifier. And maybe the invoice total/amount could be useful.

Hint / tip: There are MANY invoices. About 412. That’s a big blob of data.

In your “get all invoices” manager method, you can limit the number of invoices that you bring back.

If you want to do this, add the [Take() extension method](https://msdn.microsoft.com/en-us/library/bb300906(v=vs.110).aspx) to the end of the query statement. For example, Take(100) will return only 100 items. Nice.

**Info about the *get one* use case, for Invoice**

A common practice for *get one* use cases is to return an object that has associated data. The situation (use case, needs of the requestor, programmer preference, etc.) will determine how much data.

For this use case, for **invoice**, plan on returning the following associated data:

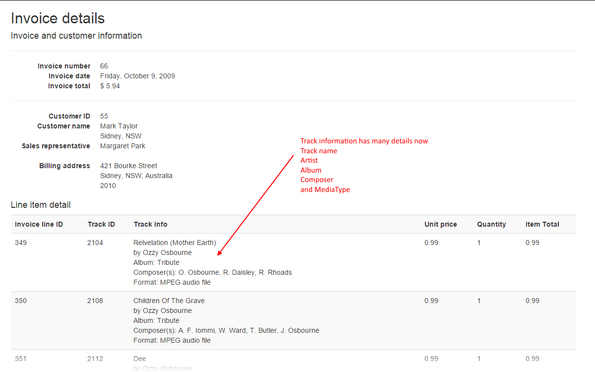
* Customer name - string
* Employee (sales rep) name - string
* The collection of invoice line items – IEnumerable<InvoiceLine>
* Each item in the collection will include…
  + Track name - string
  + Composer -string
  + Media type name - string
  + Album title - string
  + Artist name -string

Remember the design principle that you may remember from the past, when working with multiple associations:

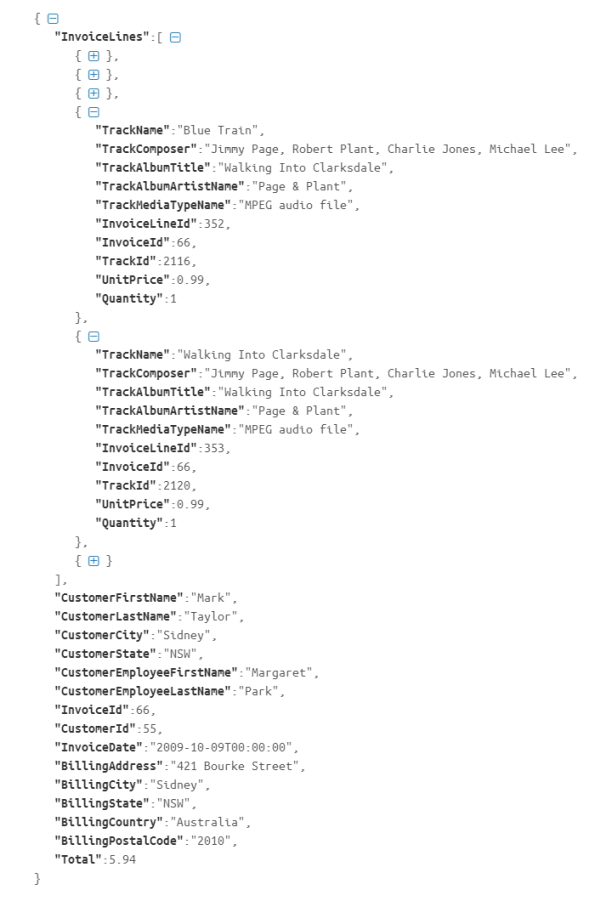
When the associated objects are sequentially connected together (as in a “chain” scenario), the string in the Include() method can be a *dot-separated string* of the class names, starting with the class name of the nearest connected object.

When the associated objects are *not* sequentially connected together (as in a “branch” scenario), add an Include() statement for *each* associated class name.

If you are looking for a visualization of what you are trying to do, here’s a screen shot of a web app that includes the data. Click to open it full-size in a new tab/window.

[](https://petermcintyre.files.wordpress.com/2016/09/invoice-detail.png)

What should the JSON data look like? Maybe like the following. Click to open it full-size in a new tab/window.

[](https://petermcintyre.files.wordpress.com/2017/09/invoice-66-json.png)

**Info about the *get one* use case, for Employee**

As noted above, a common practice for *get one* use cases is to return an object that has associated data.

For this use case, for employee, plan on returning the following associated data:

* The collection of direct-reports (the awkwardly-named Employee1 navigation property; the other employees that this employee supervises or manages)
* The employee’s supervisor/manager (the awkwardly-named Employee2 navigation property)

**Info about the *add new* use case, for Employee**

When adding a new employee object, the employee’s supervisor info (an identifier) can be provided in the data package (if you know it when adding the new employee), or not (if you do not know it, or if the added employee has no supervisor).

**Info about the *change state* use case that configures an employee’s manager**

This is a straightforward, uncomplicated command. Notice that the request’s data package must include both the identifier of the employee, and the identifier of the employee’s supervisor/manager.

Remember, in the manager method, you must validate the identifiers.

**DPS907 students:  
Info about the *change state* use case that configures an employee’s direct-reports**

The request’s data package will send the identifier of the employee (who is a supervisor/manager), and a collection of identifiers for the other employees that report to the manager/supervisor.

You must send a collection of employee identifiers.  
Question: What does an integer collection look like in JSON?

Remember, in the manager method, you must validate the identifiers.

**Before continuing…**

Make sure that the use cases above work correctly, before working with the non-text media item use cases that follow.

**Doing the work, for non-text media items**

Although the idea of working with non-text media items in a web service is well-understood and is not too difficult, it is very important to design and implement this feature carefully. We must enable the app to store non-text media items (photo, audio, pdf), and then incrementally add and test each capability.

**Enable the app to store non-text media items – Employee entity**

We must edit the Employee design model class, and add properties for the non-text media items. As you know, a media item requires two properties:

1. A string property to hold the content type
2. A byte array property to hold the content

Before doing this work, we must prepare the persistent store, by using the migrations feature that you first learned about in your web apps class. Here’s how to do this work.

Note:  
We suggest that you “test” this procedure by creating a new throw-away project.  
This tactic will enable you to test your work without affecting your assignment work.  
Then, when you’re confident, you can repeat the procedure in your assignment work.

Ensure that the app works correctly, and the use cases above have been fully implemented.

Next, enable the migrations feature, by opening the package manager console, and running this command:

|  |  |
| --- | --- |
| 1 | enable-migrations |

Then, establish a starting “before changes” point, by running these commands:

|  |  |
| --- | --- |
| 1  2 | add-migration initial -ignorechanges  update-database |

Assuming there’s no errors, and before continuing, build/compile and run the app again, and use Fiddler to test a few “get” use cases (including Employee requests), just to ensure that this work did not break anything.

Next, modify the Employee design model class. A reminder, in Solution Explorer, it’s in the Models > ChinookModelClasses folder. Add the two media item properties, and use the suggested guidance for naming them. (For the string property, use the string length data annotation to limit the length to a reasonable value. Guidance from the RFC suggests the maximum “should be” 127 characters.) Build/compile.

Activate this change with the following command. It will create a new migration class that ultimately modifies the database structure:

|  |  |
| --- | --- |
| 1  2 | add-migration employee-media-added  update-database |

Assuming there are no errors, and before continuing, build/compile and run the app again, and use Fiddler to test a few “get” use cases (including Employee requests), just to ensure that this work did not break anything.

Note:  
You can also look at the database table definitions to ensure the changes happened.  
(You may remember doing this in your web apps course.)  
In Solution Explorer, double-click the App\_Data > Chinook.mdf item, and it will open in Server Explorer.  
Navigate to the node you want, and inspect.  
When you’re done, right-click the connection, and choose “Close connection”.  
(If you don’t do this last part, it’s possible that you will have future build/compile/run problems.)

**DPS907 students:  
Enable the app to store non-text media items – Album, Track, and Invoice entities**

Modify the Album, Track, and Invoice design model classes, to include the required properties.

Activate these changes with the following command. It will create a new migration class that ultimately modifies the database structure:

|  |  |
| --- | --- |
| 1  2 | add-migration album-track-invoice-media-added  update-database |

Assuming there are no errors, and before continuing, build/compile and run the app again, and use Fiddler to test a few “get” use cases (including Invoice requests), just to ensure that this work did not affect anything.

**Use cases from above**

Here are the use cases, from above, for non-text media item interactions:

* Accept, upload, and deliver non-text internet media types (employee photos)
* Implement basic content negotiation

And, for students in DPS907, complete these additional specifications:

* Enable the use of an Album cover art image
* Enable the use of an audio Track sample
* Enable the use of a PDF version of an Invoice
* Implement beyond-basic content negotiation

**Info about the *change state (photo)* use case**

This will enable a requestor to add (or replace) a photo for a specific Employee object.

Implement this by using the command pattern.

Add photos for 5 or more employees. Use realistic photos. Do not use the same photo for all employees. If you need to, use a photo editor to crop or resize large photos.

Tip:  
Use a search engine’s “images” search feature.  
For example, use Google to search for “employee”, and then use the filter buttons to get photos of individuals (e.g. “male”, “female”).

Limit the size of the photo which can be uploaded, maybe to about 100KB. The best place to decide that is in the controller method. Return the most correct response code and error message to the requestor. Use [RFC 7231](https://tools.ietf.org/html/rfc7231) to help determine the most correct response code.

**Implement basic content negotiation**

In a new folder, probably named ServiceLayer, add a source code file to hold the media type formatter named ByteFormatter.cs. If you wish, you can copy that from the Blackboard/My.Seneca assignment upload item, the course code example repository on GitHub, or one of the Week 4 code examples. You have permission to use that source code file as-is. Make sure that you edit its namespace name, and then add it to the request-processing pipeline.

Then, modify your employee “get one” code chain to deliver the media the user requests it. As you learned in class, this requires changes to your resource models, and the manager and controller methods.

**DPS907 students:  
Info about the *change state (album cover art)* use case**

This will enable a requestor to add (or replace) an album cover image for a specific Album object.

It should be obvious that you must write some resource model classes, and a controller, for the Album entity.   
Plan on coding a “get all” method, a “get one” method that does content negotiation, and a “set/configure media item” method described next…

Implement this by using the command pattern.

Add images for 5 or more albums. Use realistic image. Do not use the same image for all albums. If you need to, use an editor to crop or resize large images.

Tip:  
Use a search engine’s “images” search feature.  
For example, use Google to search for “album cover *name-of-album*“.  
Or, use Amazon.

Limit the size of the image which can be uploaded, maybe to about 100KB. The best place to decide that is in the controller method. Return the most correct response code and error message to the requestor. Use [RFC 7231](https://tools.ietf.org/html/rfc7231) to help determine the most correct response code.

**DPS907 students:  
Info about the *change state (audio track)* use case**

This will enable a requestor to add (or replace) a short sound clip (audio track) for a specific Track object.

It should be obvious that you must write some resource model classes, and a controller, for the Track entity.   
Plan on coding a “get all” method, a “get one” method that does content negotiation, and a “set/configure media item” method described next…

Implement this by using the command pattern.

Add sound clips for 5 or more tracks. Use realistic sound clips. Do not use the same sound clips for all tracks. If you need to, use an audio editor to resample, trim, or make excerpts from large sound clips.

Limit the size of the sound clip which can be uploaded, to about 200KB. The best place to decide that is in the controller method. Return the most correct response code and error message to the requestor. Use [RFC 7231](https://tools.ietf.org/html/rfc7231) to help determine the most correct response code.

Where can you get sound clips?

1. Create them yourself
2. Download sample sounds
3. Use your own music (assuming that you have a valid license)

If you are searching for and downloading sample sounds from the web, *respect the copyright and the wishes of the sound’s creator*. Use public-domain or open-source audio if possible. Do not steal if the creator/licensor wants to be compensated.

**DPS907 students:  
Info about the *change state (PDF of invoice)* use case**

This will enable a requestor to add (or replace) a PDF for a specific Invoice object.

Implement this by using the command pattern.

Add PDFs for 5 or more invoices. Use the provided PDFs. Match the invoice number in the PDF with the identifier of the invoice that you want to work with. (The Blackboard/My.Seneca assignment upload item has a zip file with a selection of invoices.) Do not use the same PDF for all invoices.

Limit the size of the PDF which can be uploaded, maybe to about 100KB. The best place to decide that is in the controller method. Return the most correct response code and error message to the requestor. Use [RFC 7231](https://tools.ietf.org/html/rfc7231) to help determine the most correct response code.

**DPS907 students:  
Implement beyond-basic content negotiation**

This will be done for the Employee, Album, Track, and Invoice “get one” use cases.

Implement the enhanced content negotiation that you learned recently in class. It improves the quality of interaction and request-handling.

**Testing your work**

Use Fiddler.

Test all scenarios (use cases).

This time, add some requests that you know will cause HTTP response errors (400s and/or 500s), so that we know that your app is responding with the correct status code etc.

Again, remember the tip from the notes:  
Create a plain text file in your project root to store entity bodies that can be used in copy/paste.

**Saving – “exporting” – your tests**

On the left side list of requests, you can delete items that you do not want included in the export.

When you’re ready to save, choose File > Export Sessions > All Sessions…

The export format will be “HTTPArchive v1.2”. Click the Next button to choose a save location (your project’s root, in the same folder level as the “packages” and “assign1” folder) and specify a filename. Name the file by using the project name (e.g. “*whatever*.har”).

(You can test whether the export was successful. How? First, close then re-open Fiddler. Choose File > Import Sessions. Select “HTTPArchive” as the import format. Navigate to the folder that holds the “.har” file, and select it. Finally, browse through the request-response sessions.)

**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*.

When you are ready to submit your work, you will copy some of the code in your project to plain text files, so that the My.Seneca/Blackboard “SafeAssign” tool can do its job. The next section will tell you which files to copy.

From the [Blackboard web site](https://www.blackboard.com/safeassign/index.html):  
SafeAssign compares submitted assignments against a set of academic papers to identify areas of overlap between the submitted assignment and existing works.

**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. (You will be uploading a zipped version of the copy.)

3. Add a new folder named “MyCode” to the root of the copied folder.

4. Remove the “packages” folder from the copied folder; also, remove the “bin” and “obj” folders.

5. Copy these source code files from your project to the “MyCode” folder:

* Manager.cs
* Invoice\_vm.cs
* InvoicesController.cs
* Employee\_vm.cs
* EmployeesController.cs
* DPS907 students – Album\_vm.cs and AlbumsController.cs
* DPS907 students – Track\_vm.cs and TracksController.cs

For each of those files, change the “cs” file name extension to “txt”.

6. Compress/zip the copied folder. The zip file size SHOULD probably be less than 10MB (and definitely less than 50MB). If it isn’t, you haven’t followed the instructions properly.

7. Login to My.Seneca/Blackboard. Open the Web Services course area. Click the “Assignments” link on the left-side navigator. Follow the link for this assignment. 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.