Introduction

In this assignment you will create an ASP.NET web app that uses a persistent store. It is recommended you read the entire assignment before you get started.

This assignment is worth 9% of your final course grade. If you wish to submit the assignment before the due date, you can do that. If submitted after the due date, you will receive a 10% deduction for every 24-hour period the assignment is late. An assignment handed in more than five days late will receive a mark of zero.

Objective(s)

You will implement some common interaction patterns in a web app that uses a persistent store. This assignment will focus on enabling typical use cases for **Venue** objects, that is the "get all", "get one", "add new", "edit existing" and "delete existing" use cases.

It is suggested that you refer to the "Web App Project Template v1 – Part 1.pdf" and "Web App Project Template v1 – Part 2.pdf" documents to help you complete this assignment.

Specifications overview and work plan

Here is a brief list of specifications that you must implement:

- You must follow best practices.
- Implement the recommended system design guidance.
- Customize the appearance of your web app.
- Enable "get all", "get one", "add new", "edit existing" and "delete existing" use cases for the **Venue** object.
- Create the Controller and Manager classes that will work together for data service operations.

Here is a brief work plan sequence:

- Create the project, based on the project template.
- Update the NuGet packages (except for Bootstrap).
- Customize the appearance of your web app.
- Create view models and mappers that cover the use cases.
- Add methods to the Manager class to handle the use cases.
- Add a controller that interacts with the Manager object.
- Use the built-in scaffolder to create views to handle the use cases.

Getting started

Using the "**Web App Project Template S2022 V1**" project template provided by your professor, create a new web app and name it as follows: "**S2022A1**" **+ your initials**. For example, your professor would call the web app "S2022A1*NKR*". The project template is available on Blackboard.

Using the techniques that you learned in class and in the "Web App Project Template" how-to documents, update the project's code.

Build and run the web app immediately after creating the solution to ensure that you are starting with a working, error-free, base. As you write code, you should build frequently. If your project has a compilation error and you are unsure how to fix it:

- Search the internet for a solution. Sites like Stack Overflow contain a plethora of solutions to many of the common problems you may experience.
- Post on the MS Teams "Course Help" channel.
- Email your professor and include error message details and screenshots when possible.

Update the project code libraries (NuGet packages)

The web app includes several external libraries. It is a good habit to update these libraries to their latest stable versions. Unfortunately, if you update all the libraries, Visual Studio will also update Bootstrap (a popular front-end component library) and AutoMapper. Due to breaking changes between Bootstrap 3.x and 4.x, some components including the navigation bar may not look correct. If you update AutoMapper, your solution may not compile correctly. Be careful not to update these two packages.

1. Open the "NuGet Package Manager" for the solution. From the menu, choose "Tools" > "NuGet Package Manager" > "Manage NuGet Packages for Solution".
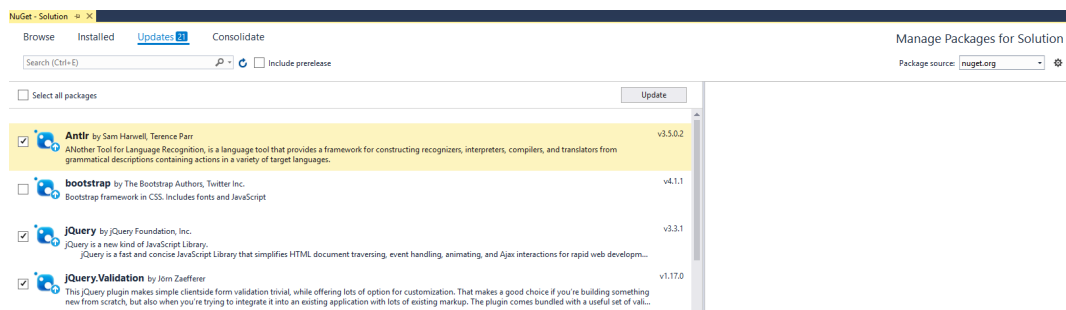
2. Change to the "Updates" tab.

3. Turn on "Select all Packages.

4. Unselect the package "bootstrap" by "The Bootstrap Authors, Twitter Inc." by removing the check next to the package name.

5. Click "Update".

6. If the "Preview Changes" window appears, click "OK".

7. If the "License Acceptance" window appears, click "I Accept".

8. You may receive a message to restart Visual Studio. If you do, now is a good time to do that.

Reminder: As you write code, you should frequently build your project. This will help you to quickly identify and fix errors. You can see a live list of errors and warnings in the Error List window.

View your web app in a browser by pressing F5 or using the menus ("Debug" > "Start").

Customize the app's appearance

You will customize the appearance of all your web apps and assignments. **Never submit an assignment that has the generic auto-generated text content.** The appearance of an app can make or break the user experience, so, part of your grade on this assignment and future assignments will take into consideration 'going the extra mile'.
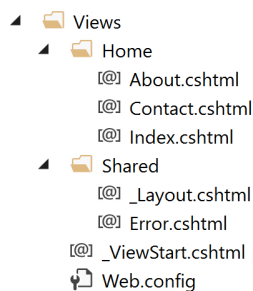
You can defer this customization work until later but do not forget to come back to it before you submit your work. Follow the customization work exactly, marks are deducted if anything is missed or not displayed as indicated.

There are four customizations you will need to make on this assignment:

1. Update the page layout
2. Edit the Home/Index view
3. Edit the Home/About view
4. Edit the Home/Contact view

CUSTOMIZATION 1: Update the page layout

1. In the Solution Explorer, expand the Views folder and then the subfolders named Home and Shared.

2. You should see the file "_Layout.cshtml". This is the layout code used for all views in the app. Think of it as a master template file.

```
▲ 📁 Views
   ▲ 📁 Home
        [@] About.cshtml
        [@] Contact.cshtml
        [@] Index.cshtml
   ▲ 📁 Shared
        [@] _Layout.cshtml
        [@] Error.cshtml
     [@] _ViewStart.cshtml
     🔧 Web.config
```

As you study the code in the layout file you will notice code expressions that begin with an "at" sign (@). Each @ symbol instructs the Razor engine to begin a C# code expression. This is similar to the PHP code expressions you have already used in the past (<? php-code ?>). You will learn more about code expressions, views, and the Razor view engine very soon.

```
<div class="container">
    <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
        </button>
        @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })
    </div>
    <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
            <li>@Html.ActionLink("Home", "Index", "Home")</li>
            <li>@Html.ActionLink("About", "About", "Home")</li>
            <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
        </ul>
    </div>
</div>
```

You will also notice the ActionLink helper. It is a HTML helper that enables the view engine to render an HTML <a> link element using static or dynamic values. It is true that you could simply write the <a …> element construct but the benefit of ActionLink is that it dynamically generates the HTML and its attributes. This benefit will become clearer when we study Attribute Routing.

3. Locate the first ActionLink, it should contain your project name as its first argument. As the text suggests, it is the name of the application and will appear in the upper-left area of the page, on the menu. Change the name of the application to "Concert Halls".

4. Update the application title. You will specify the title between the <title> </title> tags in the <head> block. Set the title so it looks like the following:

```
<title>@ViewBag.Title – Concert Halls</title>
```

5. You are about to add another menu item. When you study the code, you will see a modern navigation menu, composed as an unordered list of items (HTML "ul" and "li" elements). There are three menu items: Home, About, and Contact.

```
<li>@Html.ActionLink("Home", "Index", "Home")</li>
<li>@Html.ActionLink("About", "About", "Home")</li>
<li>@Html.ActionLink("Contact", "Contact", "Home")</li>
```

This is the visible text of the hyperlink   Action or method name   Controller name

Later in this assignment, you will create a controller. Be proactive and add a new item to the navigation menu to enable the user to easily get access to the new controller action.

As you begin typing the opening tag of the "li" element you will notice that the Visual Studio HTML Editor shows you choices that are available. This is called Intellisense.

Use the existing ActionLink statements as a template for adding the new menu item.

When you type the ActionLink method's open parenthesis, notice that another popup appears. Its purpose is to show you all the overloads available for that method so that you can select the one you want. Use the keyboard up and down arrow keys to show additional options. We want to use overload 4 of 10 where all three arguments are strings.
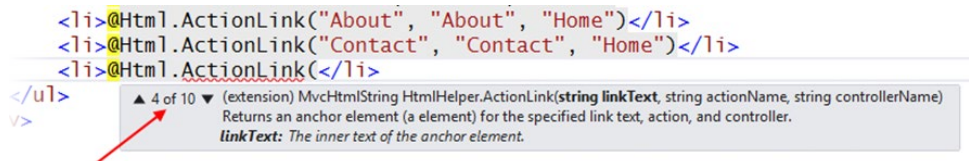

The value of linkText is the text visible to the user, set it to "Venues". The value of **actionName** is the name of the method in the controller, set it to "Index". The value of **controllerName** is the first part of the controller name, set it to "Venues". You do not need to include the word "Controller" at the end of "Venues".

6. Next you will need to edit the page footer. Change the copyright statement to read "© WEB 524 - Summer 2022 - <your first name> <your last name> @ <your Seneca username>". Your Seneca username is the first part of your email address, before "@myseneca.ca". Make sure your first and last name appear as they do on Blackboard.

7. If you build and run the project now, you will see your changes. If the navigation menu does not look that great you may have accidentally updated the Bootstrap library. You should use the NuGet package manager to revert to the latest stable release of Bootstrap 3.

CUSTOMIZATION 2: Edit the Home/Index view

1. When you debug or view your web app in a browser, the very first page that appears is the Home controller's Index view.
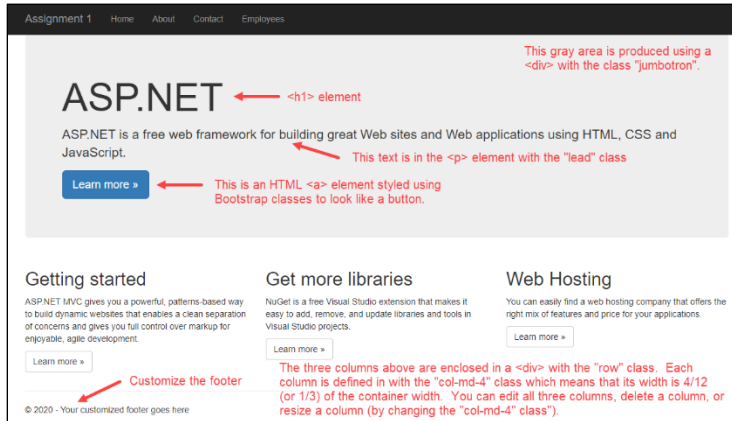
2. In Visual Studio, open the code file for the Home controller's Index view. It is a file in the Views > Home folder called Index.cshtml.

3. Edit the content of this view. At <u>minimum</u>, you must make the following suggested edits, but you can make additional edits as you please:

   a. The large title (in the HTML "h1" element) should be "Assignment 1 - <your Seneca username>".
   b. The lead text should briefly describe, in one sentence, the theme and purpose of the web app.
   c. The text in the three columns should be edited to meet your needs. You may delete the columns but if you decide to keep them, make sure you change the text. For example, you could briefly tell the user how to use the app, and how to access its functionality.

CUSTOMIZATION 3: Edit the Home/About view

1. Edit the content of the About.cshtml view file.

2. Typically, an "About" page in a web app briefly describes the theme and purpose of the web app and maybe provides some basic information about the author/programmer or company. Use your discretion to make whatever changes you feel necessary.

   Make sure you include "WEB524" and "Summer 2022" somewhere on this page. *The footer does not qualify. Each piece of information must exist on the About page itself.*

CUSTOMIZATION 4: Edit the Home/Contact view

1. Edit the content of the Contact.cshtml view file.

2. Again, use your discretion when you make your modifications. Typically, a "Contact" page in a web app describes how to contact the author/programmer or company.

   Make sure your full name appears somewhere on the page. *The footer does not qualify. Each piece of information must exist on the Contact page itself.*

Create view models and mappers that cover the use cases

In the **EntityModels** folder, study the **Venue** class. Although it will include some unfamiliar syntax, you should be able to locate and understand its properties.

In the **Models** folder, create code files to hold the venue-related **view model classes**. As you recently learned, the file name is a composite name consisting of the **singular word form** of the entity (i.e. Venue) plus the use case. You should use the suffix "ViewModel" on all view model classes.

VenueAddViewModel

We want to take advantage of *inheritance*. The first class that we should write is the "add" class, **VenueAddViewModel**.

The **VenueAddViewModel** class will be used to define the data that the user sends (using HTTP POST) from the HTML Form. The controller logic will accept the data and process it resulting (presumably) in the creation of a new Venue object in the data store. In summary, it is used to get data <u>from</u> the user <u>to</u> the web app.

Copy all properties *except the VenueId property* (with their data annotation attributes) from the **Venue** *design model class* (EntityModels/Venue.cs) then paste the properties into the **VenueAddViewModel** class you just created then fix any resulting errors. To set initial values for a property, create a constructor and add the necessary code.

Initialize a new open date to the <u>current date less 22 years</u>.

Include a **Display** attribute on the necessary properties to make the scaffolded views look nicer. Remember to do this for all view models, not just the "Add" view model. You must also include the **DataType** attribute when non-text input is expected, for example, with dates.

Add the following attribute to any date/time properties in your view model:
`[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]`

Tips

- The **DateTime** structure includes convenient methods to modify a date.
- You have learned that **DateTime.Now** will return an object that has the current date and time. Provide a negative value to the **AddYears()** method to subtract years from the current date.
- If you have correctly set the **DataType** attribute, the time component will not display in the generated view.

VenueBaseViewModel

Create a view model class to hold the base, *or basic,* properties of a venue. In almost all cases, this kind of class includes all the properties from the "add" view model class plus the **object's identifier**.

The **VenueBaseViewModel** class will be used to define the data that the web app passes to the view and displays in the web browser. In other words, it is used to get data <u>from</u> the web app <u>to</u> the user.

Use inheritance to include all the properties from the "add" class. As a reminder, the colon character (:) is used when inheriting from a Class or Interface in C#. The signature of the **VenueBaseViewModel** class would be:

```
public class VenueBaseViewModel : VenueAddViewModel { }
```

Now add the identifier property by copying it from the design model class. If the name of the **identifier property** is "**Id**", you're done, otherwise you must add the "**Key**" data annotation attribute above the identifier property.

VenueEditFormViewModel & VenueEditViewModel

The "edit existing" use case will permit the editing of the following venue properties: **Address**, **City**, **State**, **Country**, **PostalCode**, **Phone**, **Fax**, **Email**, **Website** and **OpenDate**. Do not forget, you will need the unique identifier and company name for display purposes (in the "EditForm" view model – these properties will not be modified).

In addition to the properties above, you must also support the editing of some "extra" properties. These "extra" properties will not be saved (persisted to the database), they will only exist in the view model and HTML Form. None of "extra" the properties are required and therefore an empty value must be allowed. If you place default data in the form, all default values must pass validation.

- Sign-up Password – this is a password and text must be masked as dots on the HTML Form.
- Promo Code – must contain the format "LLNNN" where "L" represents a capital letter and "N" represents a number (0-9). For example, "XY123". Ensure a user-friendly error message is displayed when the incorrect format is specified.
- Capacity – represents the number of fans that can attend a single show, it is an <u>integer between 1 and 50,000</u>.

The typical pattern for implementing the "edit existing" use case is to write two view model classes. One will carry data to the HTML Form and the other describes the data package that is posted back to the controller method from the browser. Should you use inheritance?

Don't forget to include a **Display** attribute on the necessary properties to make the scaffolded views look nicer.

Mappers for the new view models

Think about the purpose of each of the view model classes you created. Open the **Manager** class and add the necessary **AutoMapper** mappings. You may choose to defer this step until after you have added the necessary methods to the **Manager** class.

Add methods to the Manager class that handle the use cases

As a reminder, the app will implement the following use cases for the **Venue** entity: get all, get one, add new, edit existing, and delete existing.

In the **Manager.cs** code file, create method stubs for each use case. Make sure you name your methods appropriately. You can use the following information to help you code each method. Further help is available by reading the comments in the **Manager.cs** file.

Reminder: *You <u>must</u> use the **AutoMapper** library when mapping from design model classes to view model classes (and vice versa). Do not copy and paste your logic from the code examples - not all of them use the AutoMapper.*

Method Specifications

1. **Get all**

   Parameters: none
   Returns: collection of base view model objects
   Algorithm:
   a. Fetch the collection from the data store.
   b. Sort the results in ascending order by **Company**.
   c. Map the collection to a new collection of view model objects.
   d. Return the new collection.
       What type of collection is used here? Review the "collections" topic if you need a reminder.

2. **Get one**

   Parameters: the identifier (id) for the object
   Returns: base view model object, fully configured
   Algorithm:
   a. Attempt to fetch the object from the data store.
   b. If found, map the entity to a new view model object and return it.
   c. If not found, return null.

3. **Add new**

Parameters: new view model object
Returns: base view model object, fully configured
Algorithm:
    a. Attempt to add the new object; you will have to map it to a design model object.
    b. Save changes.
    c. If successful, map the added entity to a new view model object and return it.
        Why do we return a new view model object if one was already passed into the function?
    d. If unsuccessful, return null.

4. **Edit existing**

Parameters: edit view model object
Returns: base view model object, fully configured
Algorithm:
    a. Attempt to locate the object that is being edited.
    b. If not found, return null (to the controller).
    c. Otherwise, make the changes, and then save the changes.
    d. Return the freshly edited object back to the controller.

5. **Delete existing**

Parameters: the identifier (id) for the object
Returns: a Boolean representing if the record was found and deleted
Algorithm:
    a. Attempt to locate the object that is being deleted.
    b. If not found, return false (to the controller).
    c. Otherwise, remove the object, and then save the changes.
    d. Return true if everything succeeded.

Add a controller that will work with the Manager object

Create a new **Venues** controller and add a reference to the **Manager**. Notice the word "Venues" has an "s" at the end. Make sure you include the "s" in the name of your controller.

Implement the "get all" use case

You will implement the "get all" use case for a **Venue**. Fetch the collection from the **Manager** object and pass the collection to the view. This can be done in one or two lines of code.

Add a new view, using the **List** template, and the *base* view model class. At this point in time, you can test your work. Run your app, using the /venues/index URL segment.

Concert Halls	Home	About	Contact	Venues						

Venue List

Create New

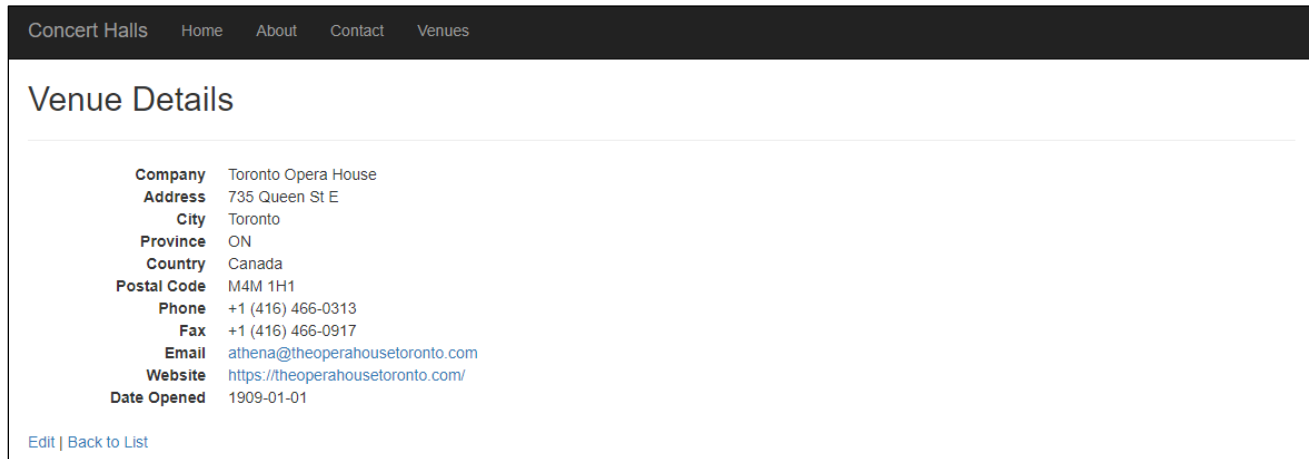| Company	Address	City	Province	Country	Postal Code	Phone	Fax	Email / Website	Date Opened	
Massey Hall	178 Victoria St	Toronto	ON	Canada	M5B 1T7	+1 (416) 872-4255		contactus@mh-rth.com https://masseyhall.mhrth.com/	1894-07-14	Edit	Details	Delete
Toronto Opera House	735 Queen St E	Toronto	ON	Canada	M4M 1H1	+1 (416) 466-0313	+1 (416) 466-0917	athena@theoperahousetoronto.com https://theoperahousetoronto.com/	1909-01-01	Edit	Details	Delete

Tips to improve the appearance of the view:

- Set the page title (that appears at the top of the browser) by modifying the **ViewBag.Title** property.
- Change titles and subtitles so they do not display the view name. They should be meaningful.
- Remove unused links or stray characters.
- Any tables or definition lists should display proper names rather than property names. For example, the **OpenDate** property could show "Date Opened"
- Because the email and website can get quite long, the screenshot above shows the two columns combined.

Implement the "get one" use case

You will implement the "get one" use case for a **Venue**. Fetch the object from the **Manager** and pass it to the view. If the object is null, return a "page not found" HTTP 404 error code.

Add a new view, using the **Details** template, and the *base* view model class. Remove any unused links and clean up the titles.

Test your work. The "Details" link on the "Venue List" page will now work.

Implement the "add new" use case

You will implement the "add new" use case for a **Venue**. The "add new" use case uses **two controller methods** as explained in the past:

The Create method with an empty parameter list will send an HTML Form to the browser. The other Create method with the "[HttpPost]" attribute will accept and process data that is posted by the user.

Create method – HTTP Get

As you have learned, the Create() method with an empty parameter list will:

- Handle a request that is made to the */venues/create* endpoint.
- Prepare data and settings needed by an HTML form.
- Display the view, which contains an HTML form.

For every view that includes an HTML form, you must decide whether the form needs **initial data** to display properly. In most cases, the answer is "**yes**" - this is a best practice. For this use case, simply create a new VenueAddViewModel object and pass it to the view.

Add a new view, using the "Create" template and the **VenueAddViewModel** class. As suggested in the past, improve the appearance of the view. Enhance the user experience, focus the cursor at the first text input field. How?

In the Create view, locate this statement:

@Html.EditorFor(model => model.Company, ...)

Add another attribute: Existing text:

...htmlAttributes = new { @class = "form-control" }...

New text:

...htmlAttributes = new { @class = "form-control", autofocus = "autofocus" }...

Create method – HTTP POST

As you have learned, the other **Create()** method will:

- Handle an HTTP POST request from an HTML form.
- Process the incoming data.
- If successful, redirect to the details view (as a confirmation to the browser user).
- If unsuccessful, require the user fix any problems and resubmit.

The "add" view model class is typically used as the parameter type, do NOT use a **FormCollection**. We want to use ASP.NET MVC *model binding*.

In the method body, the *first task* is to validate the incoming data. If not valid then return the view along with the bad data that was passed in. (The view code will then automatically display the error to the browser)

Next, attempt to create a new object by calling the appropriate method in the **Manager** object. If successful, it will return a new and fully configured object, otherwise it will return a **null**.

Finally, if successful, redirect the browser to the **Details** action (the get one use case) and ensure that you pass in the object identifier. Otherwise, return the view along with the bad data that was passed in.

Implement the "edit existing" use case

Create/modify the view files

Scaffold a view to handle the "edit existing" use case. The class notes and code examples have all the information you will need to implement this part of the work plan.

Remember: You added "extra" properties to the "edit existing" view models. These properties will be included in the HTML form and posted to the server when saving the changes. Since these properties do not match any of the design model object properties, they will be ignored. There is no need to persist the "extra" values.

After the user saves changes to a venue, redirect the browser to the **Details** action (the get one use case).

Here is an example screen capture:

Implement the "delete existing" use case

Create/modify the view files

Scaffold a view to handle the "delete existing" use case. The class notes and code examples have all the information you will need to implement this part of the work plan.

After the user deletes a venue, redirect the browser to the **Index** action (the get all use case).

Here is an example screen capture:
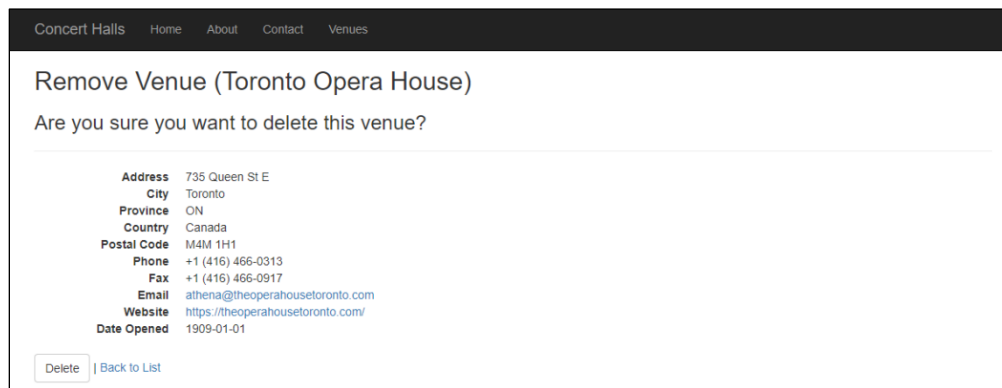
Remember …

If you have not added the **Display** attributes to your view models, your pages will include headings/titles without spaces. Go back to your view models and add the **Display** attributes to each property and rename the properties so they look much nicer. There is no need to regenerate your views after applying the attributes.

Do not include any of the default text or titles in your web app. Clean up unused links as well.

You <u>must</u> have used the provided "Web App Project Template S2022 V1" project template and **AutoMapper Instance API** for your assignment. Failure to do so will result in a huge penalty for the assignment. If you have not already done so, do not forget to customize the appearance of your web app. <u>Ensure you have not left any auto-generated text</u>.

Testing your work

Test your work by performing tasks that fulfill the use cases in the specifications.

Reminder about academic integrity

Most of the materials posted in this course are protected by copyright. It is a violation of Canada's Copyright Act and Seneca's Copyright Policy to share, post, and/or upload course material in part or in whole without the permission of the copyright owner. This includes posting materials to third-party file-sharing sites such as assignment-sharing or homework help sites. Course material includes teaching material, assignment questions, tests, and presentations created by faculty, other members of the Seneca community, or other copyright owners.

It is also prohibited to reproduce or post to a third-party commercial website work that is either your own work or the work of someone else, including (but not limited to) assignments, tests, exams, group work projects, etc. This explicit or implied intent to help others may constitute a violation of Seneca's Academic Integrity Policy and potentially involve such violations as cheating, plagiarism, contract cheating, etc.

These prohibitions remain in effect both during a student's enrollment at the college as well as withdrawal or graduation from Seneca.

This assignment must be worked on individually and you must submit your own work. You are responsible to ensure that your solution, or any part of it, is not duplicated by another student. If you choose to push your source code to a source control repository, such as GIT, ensure that you have made that repository private and have not added collaborators.

A suspected violation will be filed with the Academic Integrity Committee and may result in a grade of zero on this assignment or a failing grade in this course.

Submitting your work

Make sure you submit your assignment before the due date and time. It will take a few minutes to package up your project so make sure you give yourself a bit of time to submit the assignment.

The solution folder contains extra items that will make submission larger. The following steps will help you "clean up" unnecessary files.

1. Locate the folder that holds your solution files. You can jump to the folder using the Solution Explorer. Right-click the "Solution" item and choose "Open Folder in File Explorer".

2. Go up one level and you will see your solution folder (similar to **S2022A1NKR** but using your initials). Make a copy of your solution and change into the folder where you copied the files. For the remainder of the steps, you should be working in your copied solution!

3. Delete the "packages" folder and all its contents.

4. In the project folder (should be called **S2022A1NKR** but using your initials) contained within the solution folder, delete the "bin" and "obj" folders.

5. Compress the copied folder into a **zip** file. **Do not use 7z, RAR, or other compression algorithms (otherwise your assignment will not be marked).** The zip file should not exceed a couple of megabytes in size. If the zip file is larger than a couple of megabytes, do not submit the assignment! Please ensure you have completed all the steps correctly.

6. Login to https://my.senecacollege.ca/.

7. Open the "Web Programming Using ASP.NET" course area and click the "Assignments" link on the left-side navigator. Follow the link for this lab.

8. Submit/upload your zip file. The page will accept unlimited submissions so you may re-upload the project if you need to make changes but make sure you make all your changes before the due date. Only the last submission will be marked.