Introduction

In this assignment, we will create an app that will display lists of **Album**, **MediaType**, **Track** and **Playlist** objects. The app will also enable the user to add new **Track** objects and edit many-to-many associations while working with the **Playlist** object.

This assignment is worth 10% of your final course grade.

Specifications overview and work plan

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

- Follow best practices.
- Implement the recommended system design guidance.
- Customize the appearance of your web app *(with additional menu items)*.
- Displays lists ("get all") of **Album**, **MediaType**, **Track**, and **Playlist** objects.
- Enables "add new" functionality for **Track** objects.
- Enables "edit existing" functionality for **Playlist** objects.

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: "**S2022A3**" + your initials. For example, your professor would call the web app "S2022A3*NKR*". The project template is available on Blackboard.

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 a discussion board (on the course page in Blackboard).
- Email your professor and include error message details and screenshots when possible.

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

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. You must update the NuGet packages before you begin working on this assignment. You can refer to the instructions in Assignment 1 for more information on this topic.

Customize the app's appearance

You will customize the appearance of your web apps and assignments. **Never submit an assignment that has the generic auto-generated text content**. Please 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, including the link in the page header.

Please feel free to delete or comment out the About and Contact pages. Please also remove the link from the header.

You may also delete or comment out the three columns on the home index page. DO NOT remove the <h1> and "lead" paragraph. You must fill this in with the appropriate information.

Create view models and mappers that cover the initial use cases

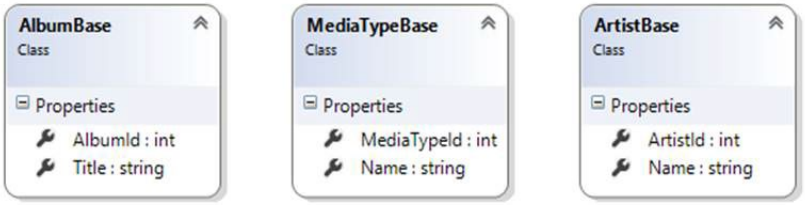
Album, MediaType and Artist entity view models

We will be working with the **Track** entity and some of its associated entities. Study the design model class diagram that was previously posted to Blackboard. It will help you visualize where the **Track** entity is located, with respect to the design model.

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

The following use cases will need view models so go ahead and write them (**AlbumBaseViewModel**, **MediaTypeBaseViewModel**, and **ArtistBaseViewModel**). None of these "…Base" classes will have navigation properties and none of these will have composed (AutoMapper flattened) properties. Keep them simple.

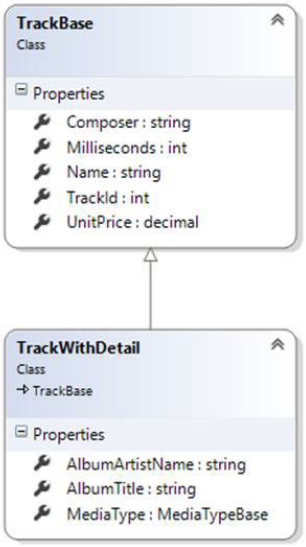
- Artist – "get all"
- Album – "get all"
- MediaType – "get all"



Track entity view model classes

For the track entity, we will support the following use cases:

- Track – "get all"
- Track – "get one"



The **TrackBaseViewModel** class needs many of the track entity's properties but not all of them – for example you can ignore the navigation properties and the **MediaTypeId** and **GenreId** properties.

You will also add a **TrackWithDetailViewModel** class. It will have a navigation property to **MediaTypeBaseViewModel**. It will also have string properties (AutoMapper flattened) for the associated album and artist descriptive data.

Later, we will need two view model classes, **TrackAddFormViewModel** and **TrackAddViewModel**, to support the use case:

- Track – "add new"

If you wish, you can create the class code blocks for them now and fill in the details later.

Playlist entity view model classes

In this assignment, you will enable basic editing of a playlist. The following image is an example of what you may end up with after you have completed the assignment.

In the image, you will see two lists of tracks. The list on the left shows all tracks, whether on the playlist or not, and includes the track name, composer, track time/length, and unit price. The list on the right shows a list of existing tracks, including the track name and unit price. The user can use the checkboxes to edit the contents of the playlist.

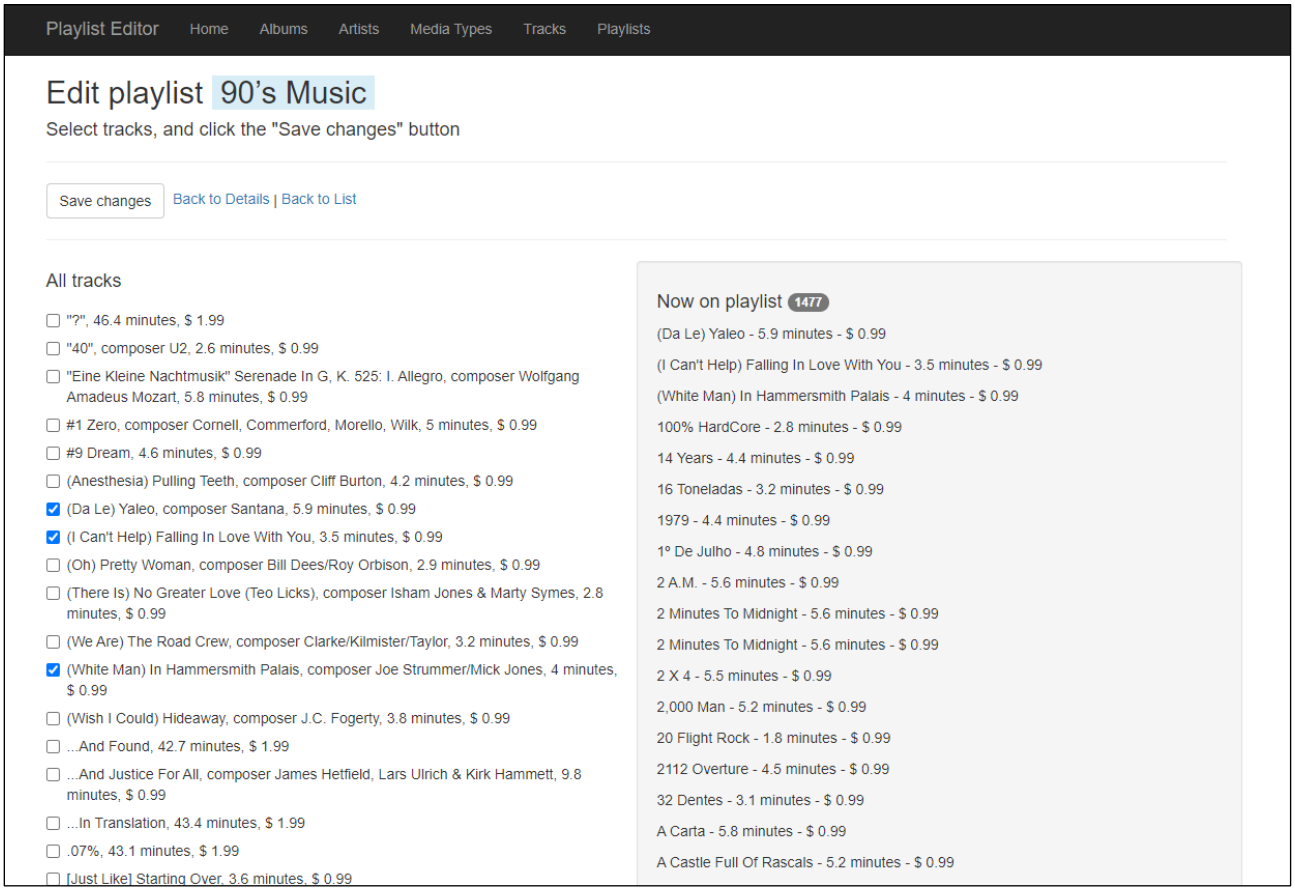
To render this view, you will need a select list object and a collection:

- The select list will be used to render the checkboxes
- The collection will hold the track objects within the playlist

You will need a **PlaylistBaseViewModel** class for displaying the playlist. The "edit existing" task will need another two view model classes:

1. A view model class named **PlaylistEditTracksFormViewModel** to hold the data that is needed to render the HTML Form.
2. A view model class named **PlaylistEditTracksViewModel** to hold the data submitted by the user.

As noted earlier, the "…Form" view model class will need a select list property (for multiple selection). It will also need another property to hold the collection of tracks that are on the current playlist. Do not forget to include the identifier in this view model class as well as text (for display purposes).



Mappers

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

Add methods to the Manager class that handle the use cases

In the Manager class, add the methods that support the use cases:

- AlbumGetAll – *Sort the results by album title.*
- ArtistGetAll – *Sort the results by artist name.*
- MediaTypeGetAll – *Sort the results by media type name.*
- TrackGetAllWithDetail – *Sort the tracks by track name.*
- TrackAdd
- PlaylistGetAll – *Sort the playlists by playlist name.*
- PlaylistGetById
- PlaylistEditTracks

For the "get all" use cases, you must use a LINQ query expression to sort the results as noted above.

For the "add new" **Track** use case, you must validate the incoming data by locating/fetching the objects that will be associated to the new **Track** object.

A new track has two associated objects, **Album** and **MediaType**. You must fetch (validate) both and configure them on the new **Track** object before saving.

You should recall that in associated-item scenarios, the "get one" method will fetch the associated object(s). Therefore, "get one" playlist must fetch its **Track** objects.

You will also need a method to handle the "edit existing" playlist method. This method will use the standard pattern for editing a to-many association of objects, as seen in the AssocManyToMany code example and in the lecture notes.

Thinking about the "edit playlist" task, you will create a page that will show or identify the playlist-being-edited. To edit/select the tracks on the playlist, the page will also have to show a list of tracks in an item-selection element. Therefore, in the Manager class, you will also need a "get all" tracks method.

Add controllers that will work with the Manager object

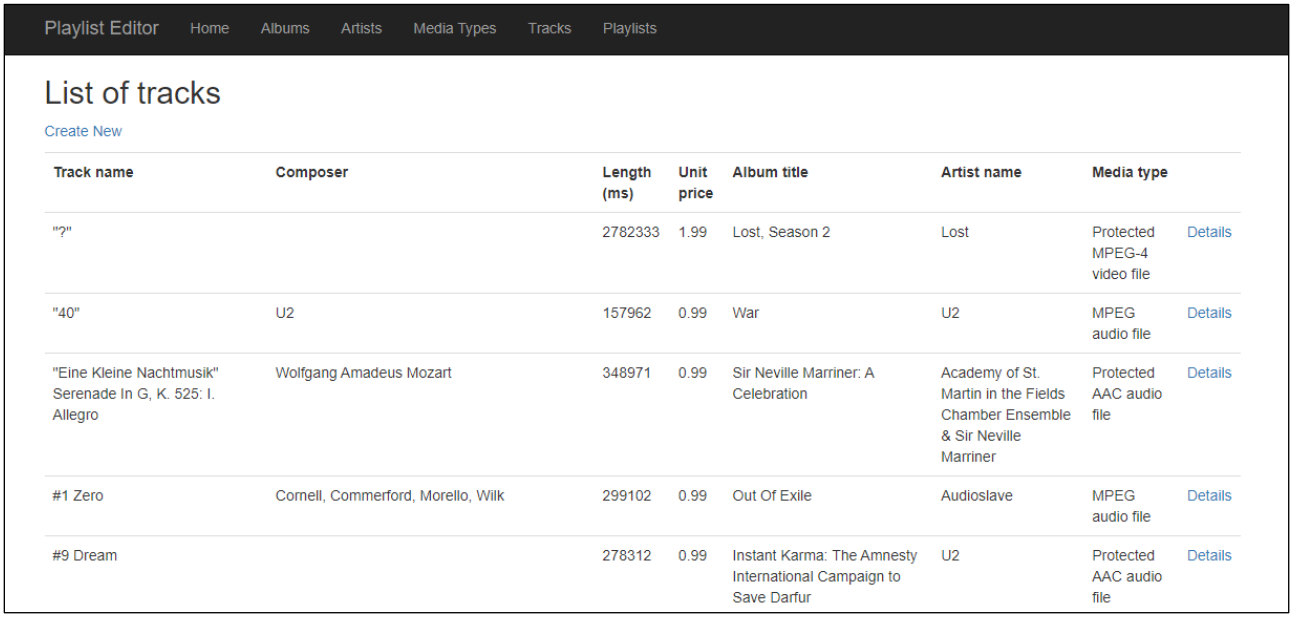
Create controllers and views to implement the "get all" use case for the **Album**, **Artist** and **MediaType** entities. They are easy to create and can help you visualize the data in those collections. Ensure you add the necessary links in the navigation bar. *Do not forget to remove unnecessary links in the scaffolded views. For example, you do not need to implement the "add new", "edit existing", "delete" or "get one" use cases so they should be removed from the scaffolded view.*

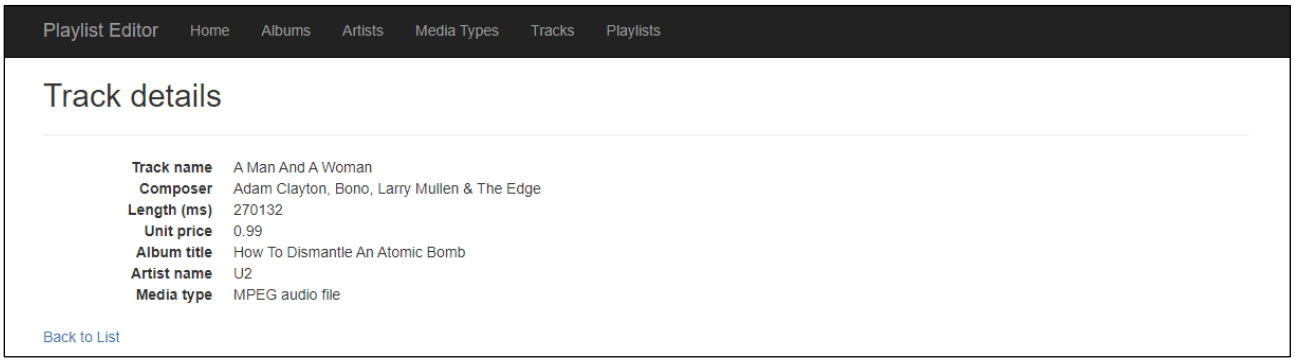
You must also create a controller for the **Track** entity. It will support the "get all" and "get one" use cases (and "add new" described in the next section). Add views for the "get all" and "get one" cases too.

Add a controller and views for the **Playlist** entity.

Checkpoint

At this point, the web app will work for “get all” tracks and “get one” track. Example images are shown on the next page.

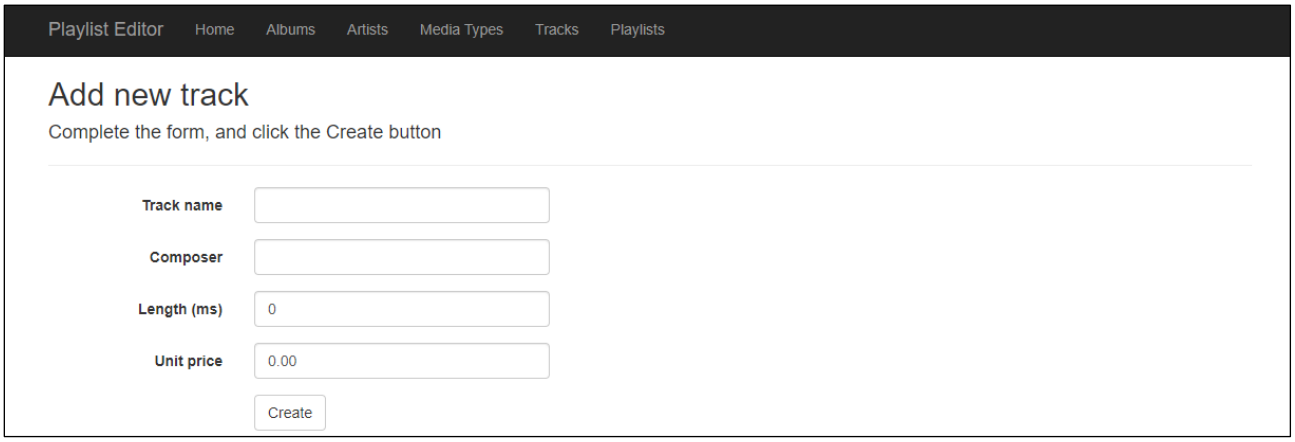




For the Track entity, implement the "add new" use case

Implement the "add new" use case for the **Track** entity. Its **TrackAddFormViewModel** class will need **SelectList** properties for both **Album** and **MediaType**. Remember to follow the naming rule for **SelectList** properties.

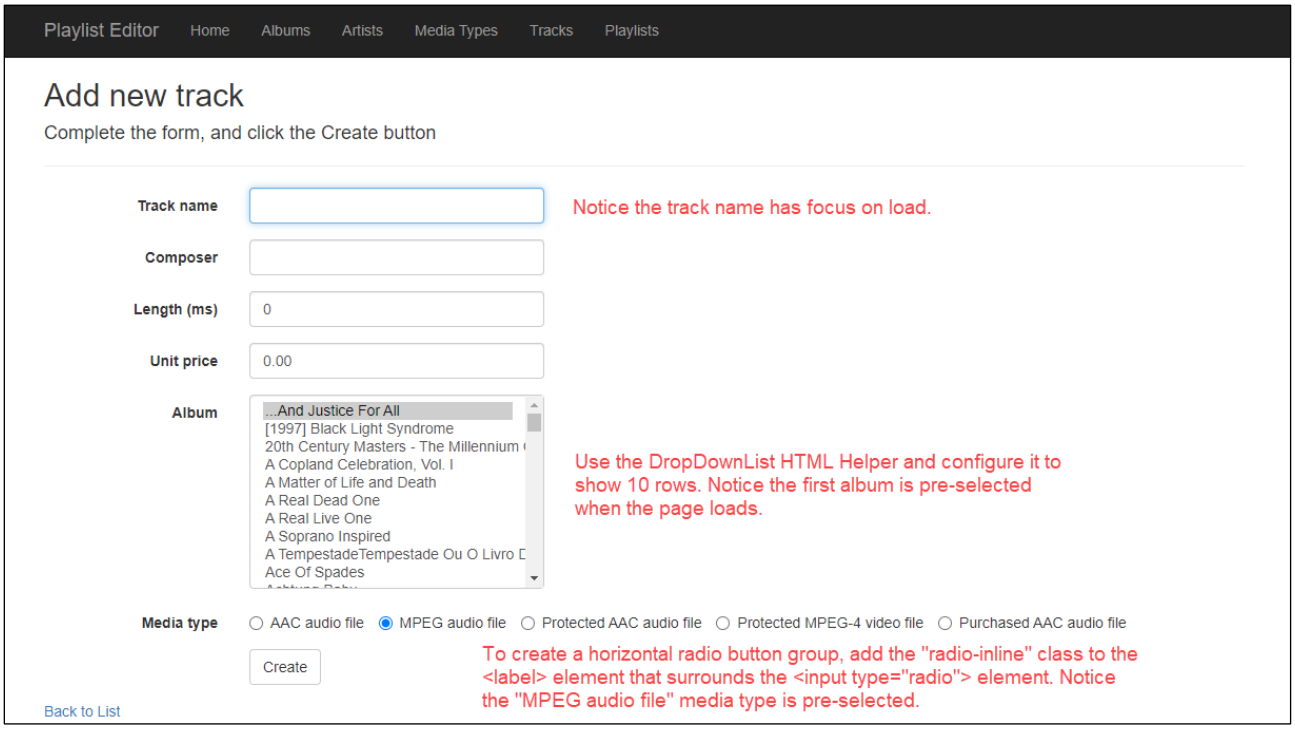
After you write the GET method (for the "add new" use case), scaffold a view. It should look something like the following:



Next, edit the view. Add the item-selection elements for the **Album** and **MediaType**. It is suggested that you use a DropDownList HTML Helper for the **Album** (make its size 10, so that it renders as a ListBox). Pre-select the first album.

Use a radio button group for the **MediaType** and ensure you render it as a horizontal radio button group. Learn how to do this by reading the Bootstrap documentation. Pre-select the "MPEG audio file" option.

It should look something like the following. Notice all controls have been styled using BootStrap, even the radio buttons. Also notice the track name is focused on page load.



TrackAddViewModel and the controller POST method

The **TrackAddViewModel** class will be like the **TrackAddFormViewModel** class however you must replace the **SelectList** properties with **int** properties named **<entity>Id**. Make them "required" by adding **[Range…]** data annotations.

Write the controller POST method next. After a successful "add new" result, redirect to the **Details** view.

For the playlist entity, implement the "get all" use cases

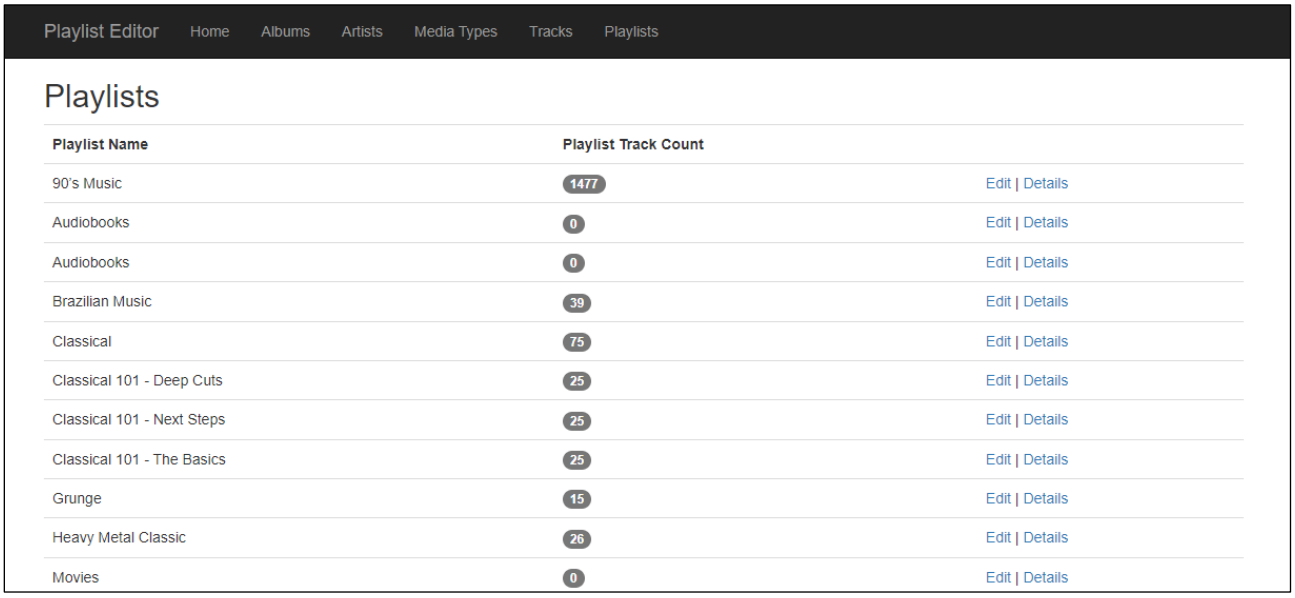
In the playlist controller, add the standard methods for the "get all" use case. Generate the "get all" view, it should look something like the following image.

How did the middle column, named "Number of tracks on this playlist", get generated? There are two changes to be made, one in the view model class and one in the Manager class.

In the **PlaylistBaseViewModel** class, add an **int** property named **TracksCount**. The name of the property is a composite name. A playlist object has a navigation property named **Tracks**, which is a collection. A collection has a property named **Count** that specifies the number of items in the collection. If named correctly, you can take advantage of the AutoMapper Flattening mechanism.

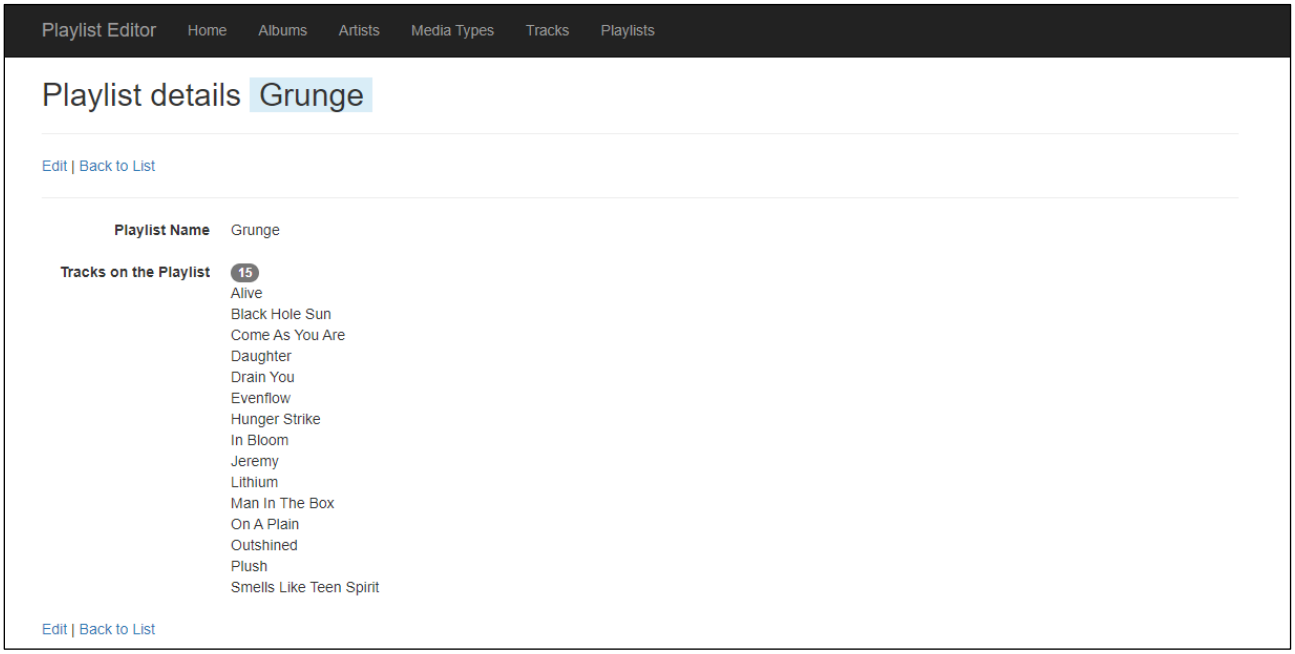
In the Manager class, edit the "get all" method so that it includes the "Tracks" property in the fetch.

Now you can render the **TracksCount** column in the view. Use a Bootstrap badge to display the number. You will see an example in the image below.



For the playlist entity, implement the "get one" use cases

In the playlist controller, add the standard methods for the "get one" use case. Generate the "get one" view, it will need hand-editing to render the collection of **Track** objects (wrapped in <p> or elements, your choice). It should look something like the following image.



Make the playlist name appear as shown in the image above (with a light blue background), then surround it with a element that uses one of the Bootstrap contextual backgrounds. Also notice the badge showing the number of tracks on the playlist.

For the playlist entity, implement the "edit existing" use cases

You are ready to implement the "edit existing" use case. If you want, you can use the pair of method stubs in the controller that the scaffolder created for the edit task. Code the rest of the methods as you have learned in your lectures and code examples.

Create a "…Form" object and configure it. In addition to the identifier property values, it will need:

- A select list object, configured with (1) all possible tracks and (2) the identifiers for the tracks that are currently associated with the playlist.
- A **TrackBaseViewModel** collection configured with the tracks that are currently associated with the playlist

Generate the view. As you already know, the scaffolder does not render collections or item-selection elements so you will need to hand-edit the view.

Make the playlist name appear as shown in the image above (with a light blue background), then surround it with a element that uses one of the Bootstrap contextual backgrounds.

In order to make a two-column layout, use the Bootstrap .row and .col-xx-xx classes. Here is how to make the container:

1. Add a element with class="row"
2. Inside this container, add two more elements, these represent both halves of the screen.
3. Each side will take half of the width or 6/12 of the grid system layout therefore add the class="col-md-6" to each container (left and right).

In the left-side container, add code for the checkbox group that displays all possible tracks. (The image is showing the result of standard <input type="checkbox"… /> elements.)

How did we render the "track name and unit price" or "track name, composer, track time/length, and unit price"? This is done by adding a read-only property (which is a property that does not have a "set" accessor) and then using that in the select list object's constructor. Here's an example of how you can add read-only properties in a view model class:

```
// Composed read-only property to display full name.
public string NameFull
{
     get
     {
          var ms = Math.Round((((double)Milliseconds / 1000) / 60), 1);

          var composer = string.IsNullOrEmpty(Composer) ? "" : ", composer " + Composer;
          var trackLength = (ms > 0) ? ", " + ms.ToString() + " minutes" : "";
          var unitPrice = (UnitPrice > 0) ? ", $ " + UnitPrice.ToString() : "";

          return string.Format("{0}{1}{2}{3}", Name, composer, trackLength, unitPrice);
     }
}

// Composed read-only property to display short name.
public string NameShort
{
     get
     {
          var ms = Math.Round((((double)Milliseconds / 1000) / 60), 1);
          var trackLength = (ms > 0) ? ms.ToString() + " minutes" : "";
          var unitPrice = (UnitPrice > 0) ? " $ " + UnitPrice.ToString() : "";

          return string.Format("{0} - {1} - {2}", Name, trackLength, unitPrice);
     }
}
```

The **NameFull** property is used for "All Tracks" in the left-side container and the **NameShort** property is used for "Now on Playlist" in the right-side container. Both containers must take exactly half of the page width so please ensure you follow the instructions above.

The right-side container will have code for the <p> elements that hold the track names now on the playlist. To make the right-side container have a grey background with rounded corners, simply add one of the Bootstrap "well" classes.

Code the method that processes the data submitted by the user

The incoming data will include the playlist identifier and an int collection (the identifiers of the tracks that will be on the playlist). Pass them on to the Manager method. A successful result will redirect to the "details" view.

Testing your work

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

Reminder about academic integrity

You must comply with Seneca College's Academic Integrity Policy. Although you may interact and collaborate with others, 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.

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 **S2022A3NKR** 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 **S2022A3NKR** 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 three submissions so you may re-upload the project if you need to make changes. Just make sure you make all your changes before the due date! Only the last submission will be marked.