Module 12: One to Many Relationships in Entity Framework

As briefly covered in the previous module Entity Framework *Code First* approach embraces Domain-Driven Design (click to read more). That is, if you are using Code First approach in Entity Framework, you are expected to first build your domain model per your application needs. You need to determine:

1. What entities (classes) need to be created,
2. What properties that these entities should have, and
3. What relationships should be set among these entities (1:1, 1:M, and M:M).

When building your model, you need to follow *Code First **conventions** and **configurations*** so that Entity Framework can read, interpret your model, and map it to a database accordingly. In other words, with these conventions and configurations, Entity Framework can extract the needed information from your model to build the corresponding database correctly.

To be able to continue with the rest of the tutorial, please create a new project. Please follow this guide to install EntityFramework into your project.

1) Conventions for Primary Key

To begin with, all classes in your model must have a **key** property, which is used by EntityFramework to create the corresponding primary key on the database side. Two conventions are provided to define a **key** property: (1) an int property named as **Id** or (2) the name of the class with **Id** suffix. If you do not follow one of these conventions to create a key property, you will receive an error, and your database will not be created (or updated).

For example, a **key** property for a class named *Album* can be either **Id** or **AlbumId** (not case-sensitive). The *key property* will be mapped to *Primary Key* field in the database with the same name (i.e., **Id** or **AlbumId**). Please add a new class (called Album) inside the Model folder (if this folder does not exist, please create it). Definition of the `Album` class is provided below:

```
public class Album
{
  public int AlbumId { get; set; }//or it could be Id
  public string AlbumName { get; set; }
  public DateTime Released { get; set; }
  public int Length { get; set; }
}
```

It is important that the key property is `int` type. In this way, the primary key values in the database will be automatically generated. The first record in the corresponding table will have 1 for its AlbumId, and for each new record this value will be incremented by one.

2) Conventions for Table and Column Names

By default, the class name will be pluralized and used to name the corresponding table in the database. For the `Album` class, the table name will be **Albums** in the database. The pluralization rules of English language are applied when naming the tables based on the class names.

Last, the column names in the tables and their types are determined based on the names of the properties in the corresponding class. Given the definition of the `Album` class, the table will have three columns (besides the primary key column): AlbumName, Released, and Length.

For now, our domain model is ready, and we will apply EntityFramework to create the backhand database. First, we need to create a new class called `MusicDbContext`. After this class file is added, please add **`using Microsoft.EntityFrameworkCore;`** statement at the top of the page. Then, inherit this class from the `DbContext` class. Last, define the `OnConfiguring` method (in which the connection string should be indicated). The `MusicDbContext` file should look like this:

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Text;

namespace Module12_home.Model
{
  class MusicDbContext: DbContext
  {
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
      optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=MusicManagement");
    }
  }
}
```

For each class for which we want a table in the database, we need to define them with `DbSet` type inside the MusicDbContext file. In this example, we need to define a `DbSet` type variable named **Albums** that will hold `Album` instances. Add the following code just before the OnConfiguring method.

```
public DbSet<Album> Albums { get; set; }
```

Open *Package Manager Console* (View → Other Windows → Package Manager Console) and run the following command: **`Add-Migration creatingDb.`** You should obtain the following screen.

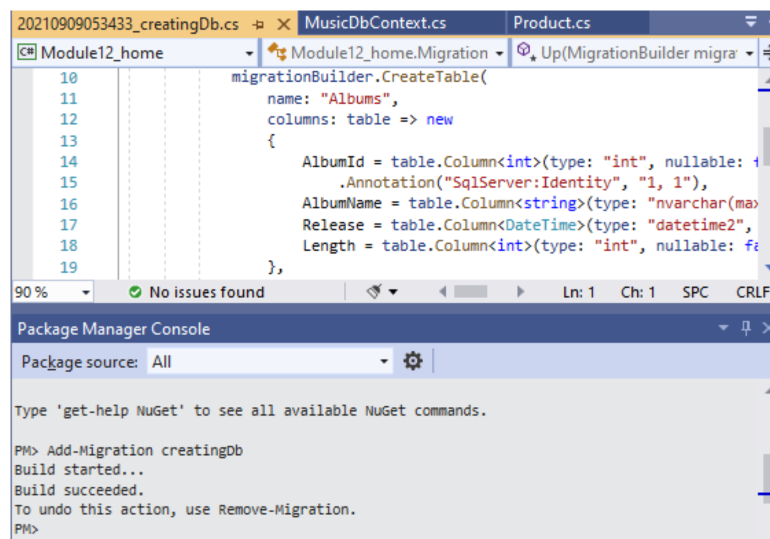

Figure 1. Adding a new migration.

Next, run the Update-Database command. You should obtain the following message in the Package Manager Console.

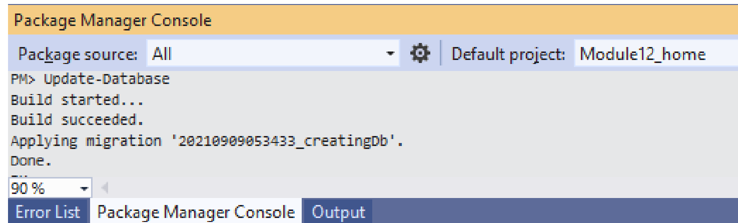

Figure 2. Applying the migration.

Now, the database is created. To see the database, please check the SQL Server Object Explorer window. The **MusicManagement** database should be available under the Databases folder, and the database should have the **Albums** table. See Figure 3.

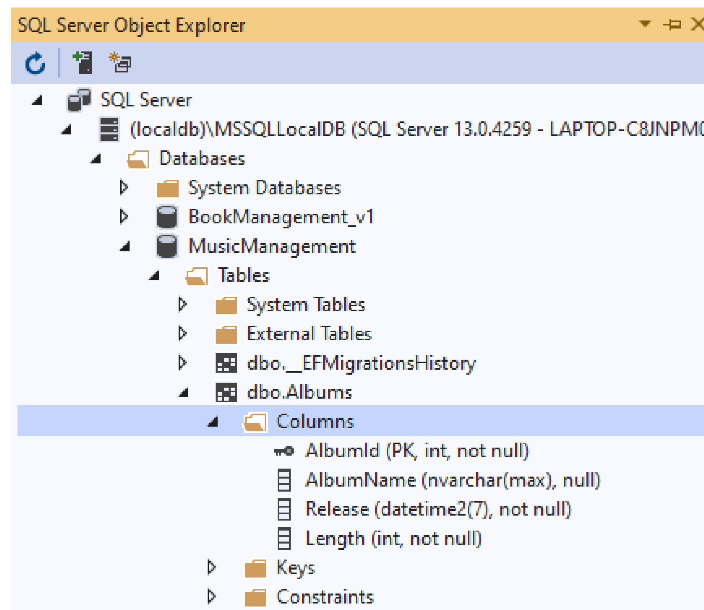

Figure 3. The MusicManagement database.

3) Data Annotations for Primary Key, Table, and Column Names

We do not have to follow the default conventions. We can override these default conventions by using **Data Annotations.** Data Annotations are applied to classes and properties directly and allow us to add some additional configurations as needed.

We will use several Data Annotations in the `Album` class to apply a different configuration. We will:

(1) Use **Table** annotation to change the default database table name,
(2) Use the **Key** annotation to define a different key property,
(3) Use the **MinLength** and **MaxLength** annotations to limit the character length for AlbumName,
(4) Use the **Column** annotation to change the table column name for the Release property,
(5) Using the **Range** annotation to ensure that Length is within a valid range.

The application of these annotations is shown in the following figure.

```
[Table("AlbumRecords")] (1)
public class Album
{
    [Key] (2)
    public int ItemId { get; set; }//or it could be Id

    [MinLength(3), MaxLength(150)] (3)
    public string AlbumName { get; set; }

    [Column("ReleaseDate")] (4)
    public DateTime Release { get; set; }

    [Range(1, 100)] (5)
    public int Length { get; set; }
}
```

Figure 4. Using data annotations to re-configure the Album class.

After any single change on your model, you are required to reflect these changes to the database by using ***migrations***. What you need to do is add a new migration. Run the **Add-Migration updatingColumns**. You should obtain the following screen (see Figure 5).

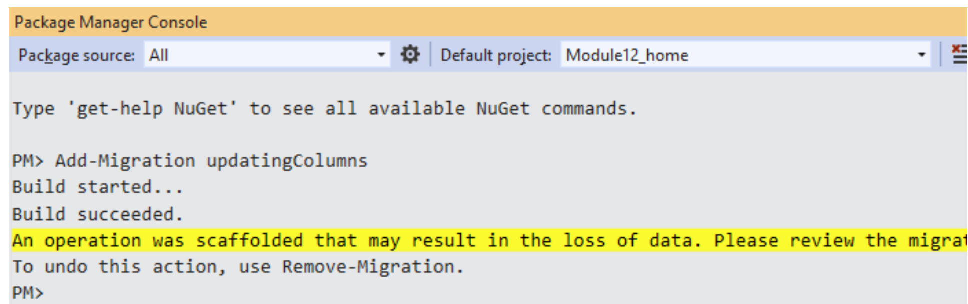

Figure 5. Running the Add-Migration command.

Next, after running the **Update-Database** command, you should get the “Done.” message in the console as shown in Figure 6.

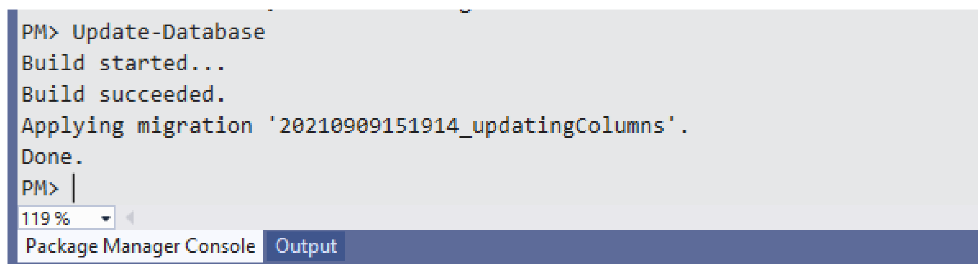

Figure 6. Running the Update-Database command.

Please view the SQL Server Object Explorer window to check the database (see Figure 7). You should notice that there are some changes in the table.

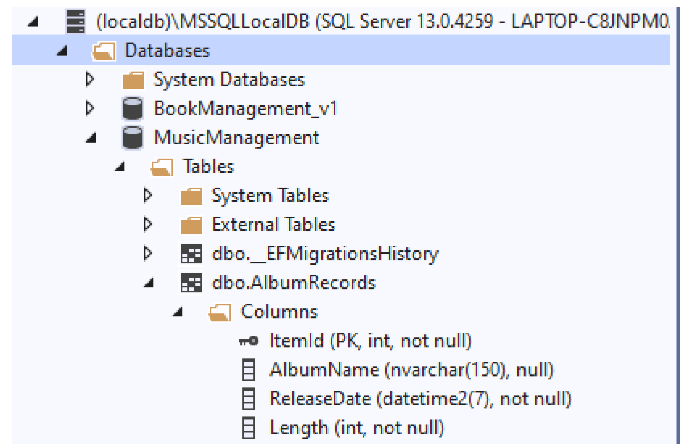

Figure 7. The updated version of the MusicManagement database.

First, the table name is changed to AlbumRecords, the primary key is changed to ItemId, and the Release column name is changed to ReleaseDate. The configuration about the minimum and maximum length for AlbumName was not reflected in the database since in SQL there is no such option. However, if we intend to add an album name with less than 3 and more than 50 characters the compiler should thrown a validation error even before interacting with the database. Similarly, the range for a Length was not applied in the database. However, if the users attempt to any invalid range for the Length of an album, an error would be thrown, and the record could not be added to the database.

4) A Brief Look at 1:M Relationships

1:M relationship is the most common relationship type in database design, since it happens a lot in real world (a customer can have many credit cards, but each credit card is assigned to a single person, or one professor (adviser) has many advisees, but one advisee has only one adviser, etc.)

In 1:M relationships, a single record from the parent table (e.g., adviser) can possibly relate to one or more child records in another table (e.g., advisee). On the other hand, the child records HAVE to be associated with a single parent record (it CANNOT be more than 1).

Below is an example table structure for a 1:M relationship between *Customer* and *CreditCard*. *CreditCard* is the dependent (or child) entity in this case, which means that it has to have a **foreign key** field (OwnerId) that indeed references the *CustomerId* field in the parent table (i.e., Customer).

CUSTOMER

CustomerId	Firstname	Lastname
22	John	Marshmellow
23	Bobby	Dixon
24	Clark	Kent

CREDITCARD

CardId	CardNumber	ExpireOn	OwnerId
144	123456789	8/8/2011	23
145	987654321	1/7/2016	23
146	456789321	2/2/2018	23
147	321654987	3/3/2020	24

5) One-to-Many (1:M) Relationships in EntityFramework

In our application, so far, we have created only `Album` class. We will add a new entity, called Song. Please create a new class inside the *Model* folder and name it `Song`. The Song class will have the following properties as shown in Figure 8.

```
public class Song
{
    public int Id { get; set; }

    [MinLength(2), MaxLength(50)]
    public string Title { get; set; }

    [Range(30, 600)]
    public int Length { get; set; }
}
```

Figure 8. Definition of the Song class.

We want to create a table in the database for the Song class. To tell this to EntityFramework, add the following line to the DbContext file:

```
public DbSet<Song> Songs { get; set; }
```

There is some kind of relationship between `Album` and `Song`: An album can have many songs, although a song can belong to a single album, which is a great example for 1:M relationship. However, at the moment, these two entities are not associated, and they know nothing about each other. To indicate this relationship among entities in the domain model, we will use ***Navigation*** properties.

Navigation properties should be added in both classes. For the `Album` class, the navigation property is simple. It is just a `List` type property that can hold `Song` objects. This property indicates that an album can have many songs. The updated definition of the `Album` class is provided in Figure 9.

```
public class Album
{
    public int Id { get; set; }

    [MinLength(3), MaxLength(150)]
    public string AlbumName { get; set; }

    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    public int Length { get; set; }

    //Navigation property to indicate the child entity
    //An album can have many songs
    public List<Song> Songs { get; set; }
}
```

Figure 9. The updated definition of the Album class.

Right now, `Album` class is aware that it has some relationship with the `Song` class. However, the `Song` class does not know about the `Album` class yet. We need to add the navigation property to the `Song` class.

The navigation property for the Song class is basically a Song type property called `Album`. This property indicates that a song can only one album. The updated class definition is shown in Figure 10.

```
public class Song
{
    public int Id { get; set; }

    [MinLength(2), MaxLength(50)]
    public string Title { get; set; }

    [Range(30, 600)]
    public int Length { get; set; }

    //Navigation property to indicate the parent entity
    //A song can have ONLY ONE album
    public Album ParentAlbum { get; set; }
}
```

Figure 10. The updated definition of the Song class.

Please add a new migration by executing Add-Migration command and then update the database by executing the Update-Database command in the Package Manager Console. After executing these commands, please view the database (see Figure 11). You should notice several changes. The most important change is the **ParentAlbumId** column added to the Songs table. You may notice that this column is indicated with a key icon. This icon tells us that this is a **foreign key** field .

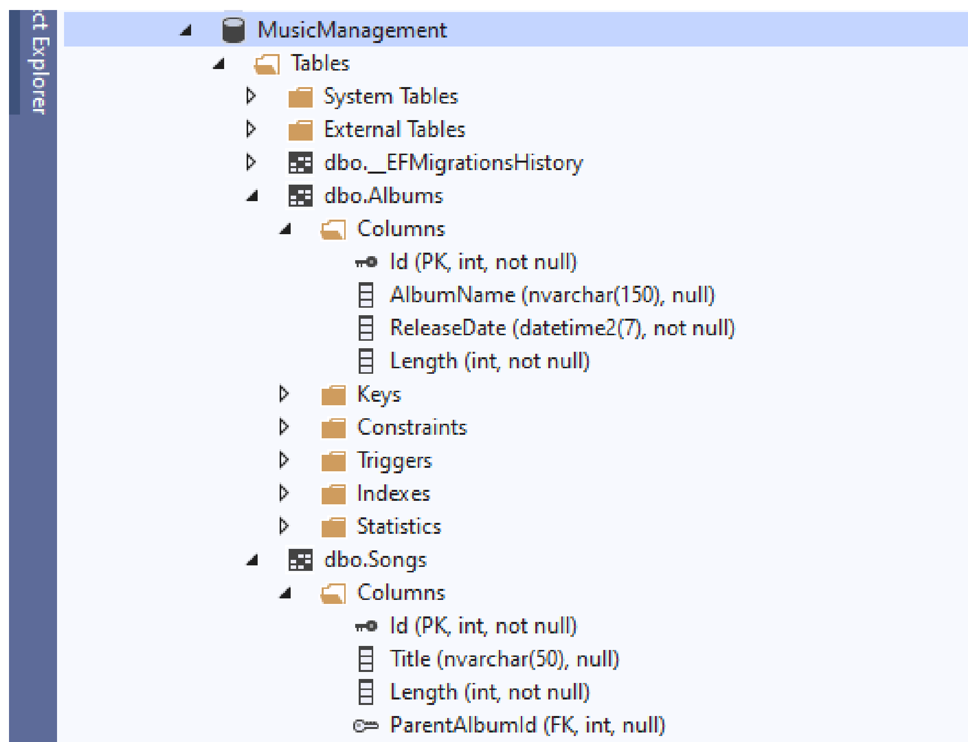

Figure 11. Songs table has a foreign key called ParentAlbumId.

Thanks to the navigation properties, EntityFramework Code First was able to infer that there is a 1:M relationship between the Song and Album entities, where Song is the child or dependent and Album is the parent. It, then, created a foreign key in the database using the pattern [Name of navigation property][Primary Key of related class] (i.e., ParentAlbumId). Also note that the foreign key is nullable, meaning that a song does not have to be associated with an album.

If you right click on the Songs table and then choose View Designer, you can view the definition of the table (see Figure 12). You can see that ParentAlbumId references the Id column of the Albums table.

Name	Data Type	Allow Nulls	Default
Id	int	☐	
Title	nvarchar(50)	☑	
Length	int	☐	
ParentAlbumId	int	☑	
		☐	

Keys (1)
PK_Songs (Primary Key, Clustered: Id)
Check Constraints (0)
Indexes (1)
IX_Songs_ParentAlbumId (ParentAlbumId)
Foreign Keys (1)
FK_Songs_Albums_ParentAlbumId (Id)
Triggers (0)

```
1  TE TABLE [dbo].[Songs] (
2  [Id]            INT           IDENTITY (1, 1) NOT NULL,
3  [Title]         NVARCHAR (50) NULL,
4  [Length]        INT           NOT NULL,
5  [ParentAlbumId] INT           NULL,
6  CONSTRAINT [PK_Songs] PRIMARY KEY CLUSTERED ([Id] ASC),
7  CONSTRAINT [FK_Songs_Albums_ParentAlbumId] FOREIGN KEY ([ParentAlbumId]) REFERENCES [dbo].[Albums] ([Id])
8
```

Figure 12. Definition of the Songs table.

Although EntityFramework creates the foreign key automatically, it is a very common practice to indicate this foreign key explicitly through navigation properties. To this this, we create a property, named `AlbumId`, which will serve as the **foreign key** for this 1:M relationship. We have to add the `ForeignKey` annotation to indicate that this property is a foreign key. If we don't do this, EntityFramework creates a foreign key itself (named ParentAlbumId) as it did above, and additionally, it creates a column named AlbumId, which is NOT what we want. We want **AlbumId** to be the foreign key.

```
public class Song
{
    public int Id { get; set; }

    [MinLength(2), MaxLength(50)]
    public string Title { get; set; }

    [Range(30, 600)]
    public int Length { get; set; }

    //Navigation property to indicate the parent entity
    //A song can have ONLY ONE album
    public Album ParentAlbum { get; set; }

    //Defining the foreign key
    [ForeignKey("ParentAlbum")]
    public int AlbumId { get; set; }
}
```

Figure 13. Adding a foreign key field to the Song class.

Please add a new migration by executing Add-Migration command and then update the database by executing the Update-Database command in the Package Manager Console. After executing these commands, please view the database (see Figure 14). You should notice that the foreign key is now **AlbumId** column as we determined through the navigation properties.

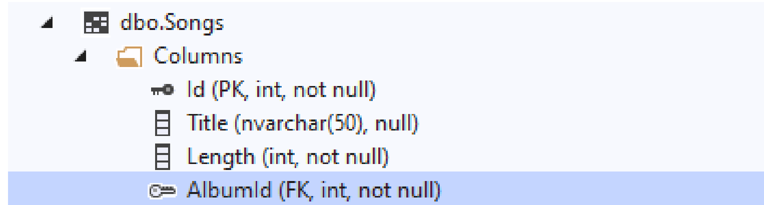

Figure 14. Updated foreign key of the Song class.

We will do a one little change before we complete finalizing the domain model. As you remember, the Song objects have a length property. By summing up the length of all songs in an album, we can compute the album length. In other words, album length depends on the length of the songs that it contains. Therefore, it should be computed rather than allowing users to enter it manually, which may cause inconsistencies if invalid data is entered.

We will keep the **Length** property in the class definition, but we do not want to create a column in the database to store the length information for albums. To achieve this, we need to add the `NotMapped` annotation just before the Length property, as shown in Figure 15 below.

```
public class Album
{
    public int Id { get; set; }

    [MinLength(3), MaxLength(150)]
    public string AlbumName { get; set; }

    public DateTime ReleaseDate { get; set; }

    [NotMapped]
    public int Length { get; set; }

    //Navigation property to indicate the child entity
    //An album can have many songs
    public List<Song> Songs { get; set; }
}
```

Figure 15. Adding the NotMapped annotation.

After this change in the domain model, execute the Add-Migration and Update-Database commands in the Package Manager Console. After executing these commands, please view the database. You should notice that there is no Length column in the Albums table anymore.

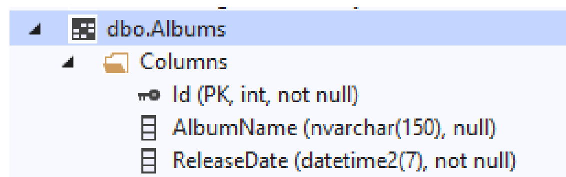

Figure 16. Length property is dropped from the Albums table.

5) Creating and listing database records in 1:M relationship

Our domain is ready and now we are ready to build the application to manage the albums and songs. In the application interface, we will add a Tab control and inside there we will add three tab-pages, as shown in Figure 17 below. First, please build the tab-page for creating an album. While we use a textbox (`txt_albumName`) to allow users enter an album name, we use the **DateTimePicker** control (`dtp_releaseDate`) to let them choose the release date through a calendar shown in a popup window.

Figure 17. Manage Albums tab page.

Please double click on the Create button to create its click event handler (see Figure 18). We will add try-catch block inside the click event handler to catch any possible error in the code (e.g., failed database connection). As indicated with three lines of comments, the code here can be grouped into three parts.

```
private void btn_createAlbum_Click(object sender, EventArgs e)
{
    try
    {
        //Create an instance of Album class

        //Add the album instance to the database

        //Show message and reset the fields

    }
    catch (Exception ex)
    {
        lbl_albumMessage.Text = "An error has occured. More info: " + ex.Message;
    }
}
```

Figure 18. Click event handler of the Create button.

The first part is relatively simpler. First, a new Album object is created (named album). Then, we set the value of `AlbumName` to the *Text* property of the txt_albumName and the value of `ReleaseDate` to the *Value* property of dtp_yearReleased. The complete code is shown in Figure 19 below.

```
//Create an instance of Album class
Album album = new Album();
album.AlbumName = txt_albumName.Text;
album.ReleaseDate = dtp_yearReleased.Value;

//Add the album instance to the database

//Show message and reset the fields
```

Figure 19. Creating an instance of Album.

In the second part, we create an instance of the `MusicDbContext` class, called `dbContext`. We use `dbContext` to add the `album` to the `Albums`. Remember that `Albums` is a `DbSet` type, which is mapped to the Albums table in the database. Last, we need to call the `SaveChanges` method to apply the changes. The complete code is shown in Figure 20.

```
//Add the album instance to the database
MusicDbContext dbContext = new MusicDbContext();
dbContext.Albums.Add(album);
dbContext.SaveChanges();
```

Figure 20. Recording the album item to the database.

In the last part, we will display and format the success message and reset the fields. See Figure 21.

```
//Show message and reset the fields
lbl_albumMessage.Text = "Album is successfully created.";
lbl_albumMessage.ForeColor = Color.DarkOliveGreen;
txt_albumName.Text = "";
dtp_yearReleased.Value = DateTime.Now;
```

Figure 21. Displaying the success message and resetting the fields.

Run your application (press F5 key) and then fill out the form to create an album, as shown in Figure 22.

Figure 22. Entering sample album data.

After pressing the Create button, the album should be recorded in the database, and you should see the screen shown in Figure 23.

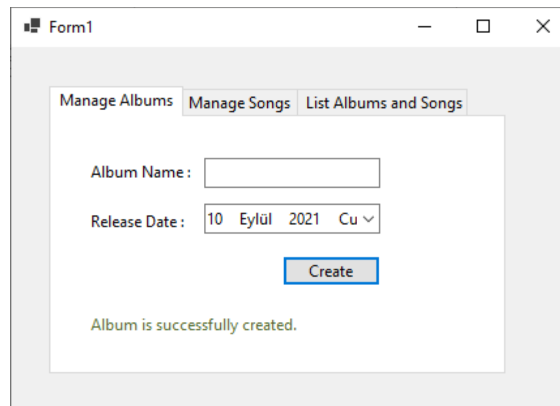

Figure 23. Adding the album to the database.

Before continuing, please define the **dbContext** variable at class level so that it can be used for all database operations throughout the application. Now that we are able to create new album records, we can display them in the third tab page as shown in Figure 24. In this tab page, there are two listbox controls next to each other and a button to show the songs for the selected album.

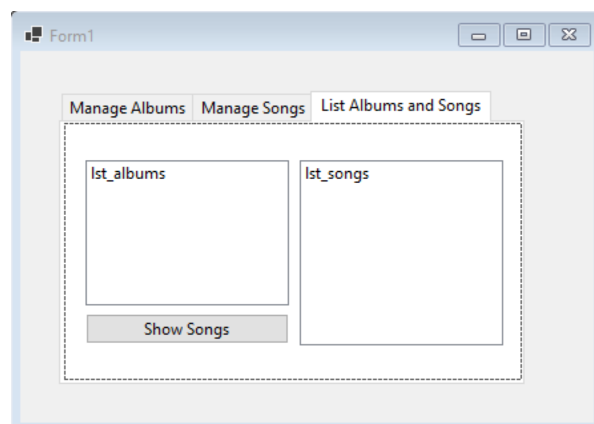

Figure 24. The interface of the third tab page.

Our goal is to display the albums when the third tab (List Albums and Songs) is selected. Therefore, we need an event handler that should be triggered when the selected tab is changed. For this purpose, we can use the **SelectedIndexChanged** event. There is no automatic way to create this event handler and we need to define it manually using the code shown in Figure 25.

```
private void tabControl1_SelectedIndexChanged(object sender, EventArgs e)
{

}
```

Figure 25. SelectedIndexChanged event handler for the tab control.

Since we created this event handler method by ourselves, we need to bind this method to the tab control manually. Please select the tab control in the design view, and then choose the Events tab in the Properties

window (see Figure 26). For the **SelectedIndexChanged** event, please select the method that we have just created above.

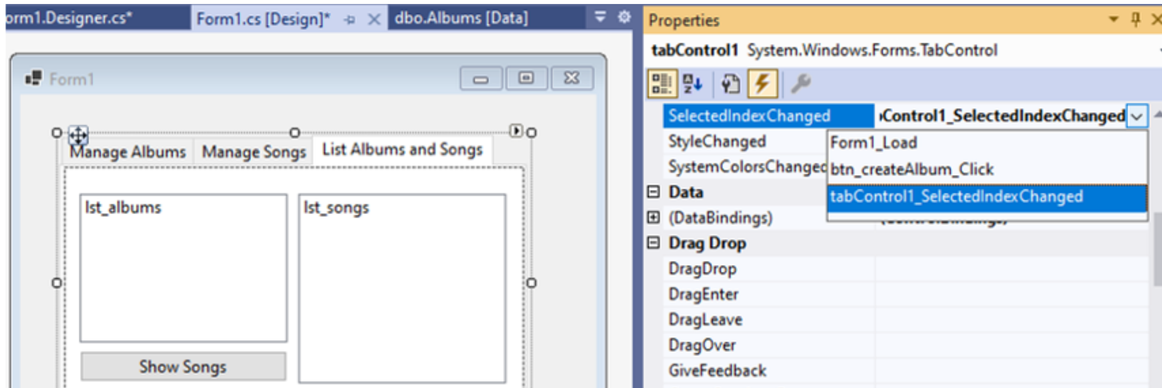

Figure 26. Setting the handler method for the SelectedIndexChanged event.

Inside the SelectedIndexChanged event handler, we need to check if the selected tab page is the page where the albums and songs are listed (i.e., the third tab page). If so, we fetch the albums from the database (using `dbContext`) and then bound them to the listbox by using the `DataSource` property. The ValueMember and DisplayMember properties should be also set properly. The complete code that goes in to the SelectedIndexChanged event handler is shown in Figure 27 below.

```
private void tabControl1_SelectedIndexChanged(object sender, EventArgs e)
{
    if(tabControl1.SelectedTab == tabpage_albumsSongs)
    {
        lst_albums.DataSource = dbContext.Albums.ToList();
        lst_albums.ValueMember = "Id";
        lst_albums.DisplayMember = "AlbumName";
    }
}
```

Figure 27. Implementation of the SelectedIndexChanged event handler.

Please run your application and add a new Album. The program should switch to the third tab page, and the existing albums should be populated in the listbox as shown in Figure 28 below.

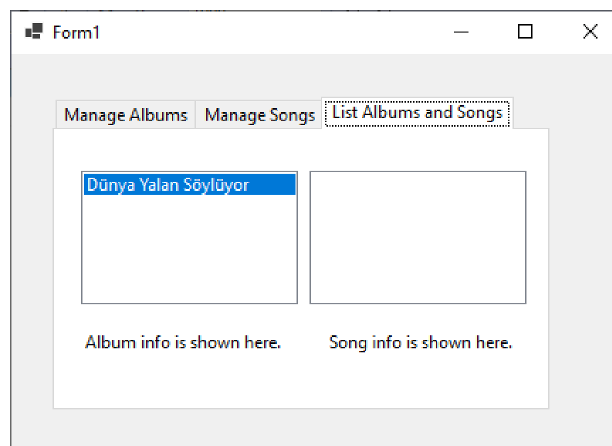

Figure 28. The existing albums are listed properly in the third tab page.

Next, we will implement the Songs tab page where uses should be able to create new songs. The interface for tis tab page is shown in Figure 28. Besides two text fields to allow users to enter the title and the length of the song to be created, there is a *ComboBox* control (named `comboBox_albums`) to allow users to select the album to which the song belongs.

Figure 29. The existing albums are listed properly in the third tab page.

This ComboBox should list the existing albums. That is, when the Manage Songs tab page is selected, we need to get the album records from the table and bind them to the ComboBox so that users can make a proper album selection when adding a new song.

This task is very similar to what we have done just above to display the existing albums when the third tab page is selected. Following the same logic, please write the code to bind the existing albums to the ComboBox when the second tab page (i.e., Manage Songs) is selected (see Figure 30).

```
private void tabControl1_SelectedIndexChanged(object sender, EventArgs e)
{
    if(tabControl1.SelectedTab == tabpage_albumsSongs)
    {
        lst_albums.DataSource = dbContext.Albums.ToList();
        lst_albums.ValueMember = "Id";
        lst_albums.DisplayMember = "AlbumName";
    }

    //Use else if to bind albums to combobox if tabpage_songs is selected

}
```

Figure 30. Binding the existing albums to the ComboBox in the second tab page.

Please run your application and select the Manage Songs tab page. The albus should be listed in the ComboBox as shown in Figure 31 below.

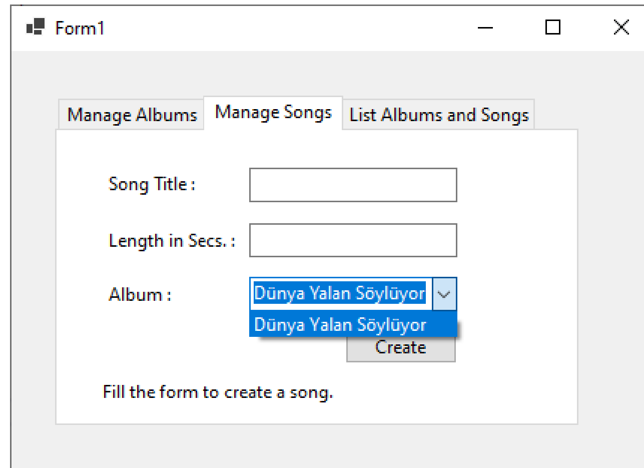

Figure 31. Existing albums are listed in the ComboBox.

Now that the albums are available for users to select for a new song, we can implement the click event handler for the Create button so that users can create a new song. Please double click on the Create button and add try-catch statements inside the click event handler as shown in Figure 32 below.

```
private void btn_createSong_Click(object sender, EventArgs e)
{
    try
    {
        //Create an instance of Song class


        //Add the song instance to the database


        //Show message and reset the fields


    }
    catch (Exception ex)
    {
        lbl_songMessage.Text = "An error has occured. More info: " + ex.Message;
    }
}
```

Figure 32. Click event handler for creating songs.

Similar to the click event handler for creating an album, the code for creating a song is basically composed of three parts, as indicated through comments in Figure 32.

In the first part, a new Song object is created (named song) and its properties are properly set to the corresponding user entries. Differently, here we need to get the id of the selected album from the ComboBox and assign it to the AlbumId property. The complete code is shown in Figure 33 below.

```
//Create an instance of Song class
Song song = new Song();
song.Title= txt_songTitle.Text;
song.Length = int.Parse(txt_songLength.Text);
song.AlbumId = (int)comboBox_albums.SelectedValue;

//Add the song instance to the database

//Show message and reset the fields
```
Figure 33. Creating an instance of Album.

In the second part, we need to use `dbContext` to add the `song` to the `Songs`. Remember that `Songs` is a `DbSet` type, which is mapped to the Songs table in the database. Last, we need to call the `SaveChanges` method to apply the changes. The complete code is shown in Figure 34.

```
//Add the song instance to the database
dbContext.Songs.Add(song);
dbContext.SaveChanges();
```
Figure 34. Recording the song item to the database.

In the last part, we will display and format the success message and reset the fields. See Figure 35.

```
//Show message and reset the fields
lbl_songMessage.Text = "Song is successfully created.";
lbl_songMessage.ForeColor = Color.DarkOliveGreen;
txt_songTitle.Text = "";
txt_songLength.Text = "";
```
Figure 35. Displaying the success message and resetting the fields.

Run your application (press F5 key) and then fill out the form to create a song. Sample screens are shown in Figure 36.

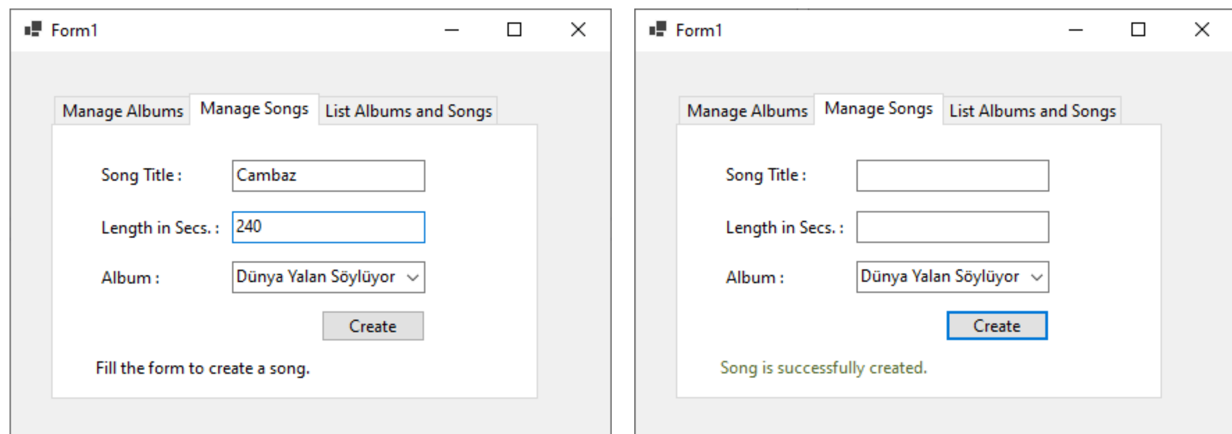

Figure 36. Adding a sample song in runtime.

We are about to complete the application. You may remember that in the third tab page, the existing albums are shown in a listbox (see Figure 37). In this page, there is a missing functionality: when an album is selected, the songs belonging to that album should be displayed in the listbox on the right-hand side.

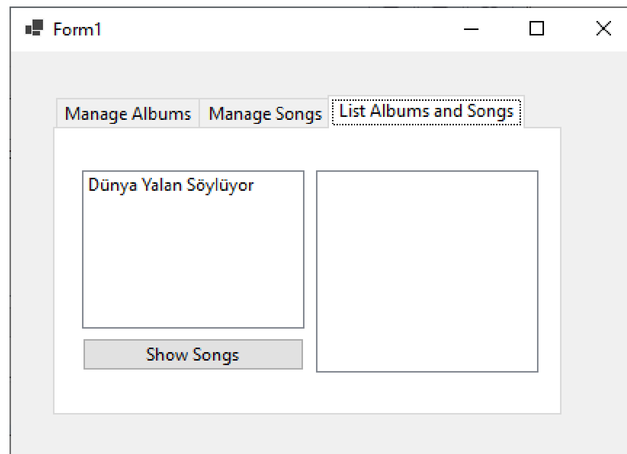

Figure 37. The design of the third tab page.

To implement this functionality, please double click on the Show Songs button, which should create its click event handler (see Figure 38). What we will do is to get the id of the selected album and use this value to fetch from the database only the songs that belong to the indicated album.

The code shown below will fetch all songs from the database. However, we do not want to show all of them.

```
private void btn_showSongs_Click(object sender, EventArgs e)
{
    int selectedAlbumId = (int)lst_albums.SelectedValue;

    List<Song> filteredSongs = dbContext.Songs.ToList();
}
```

Figure 38. The click event handler for the Show Songs button.

We will use **Where** to keep only the songs that belong to the selected album. Each song has an `AlbumId` property. We can filter the results by checking if the `AlbumId` value is equal to the Id of the selected album (i.e., `selectedAlbumId`). To achieve this, we will write a lambda expression inside **Where,** as shown in Figure 39 below.

```
int selectedAlbumId = (int)lst_albums.SelectedValue;

List<Song> filteredSongs = dbContext.Songs.Where(s => s.AlbumId == selectedAlbumId).ToList();
```

Figure 39. Filtering the songs based on the selectedAlbumId.

Next, bind the `filteredSongs` to `lst_songs`, as shown in Figure 40. We set the ValueMember to Id and DisplayMember to Title. Do you have any guess where do **Id** and **Title** values come from?

```
lst_songs.DataSource = filteredSongs;
lst_songs.ValueMember = "Id";
lst_songs.DisplayMember = "Title";
```

Figure 40. Binding filteredSongs to the listbox.

Run your application and test it. The songs for the selected album should be displayed as shown in Figure 41 below.

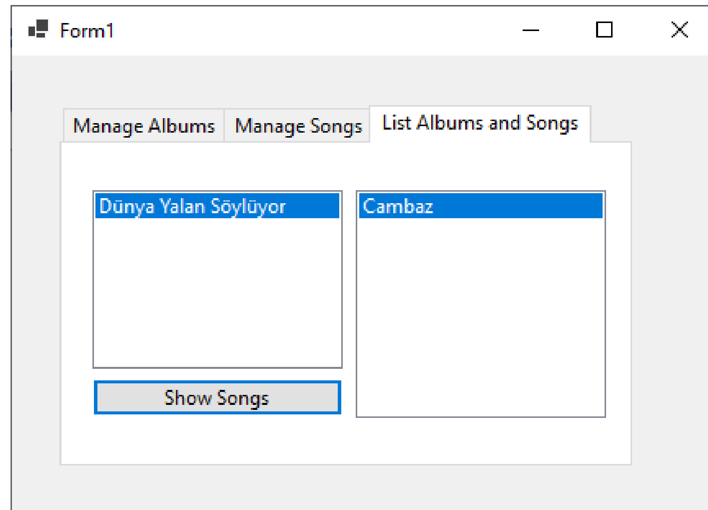

Figure 41. Songs are displayed for the selected album.