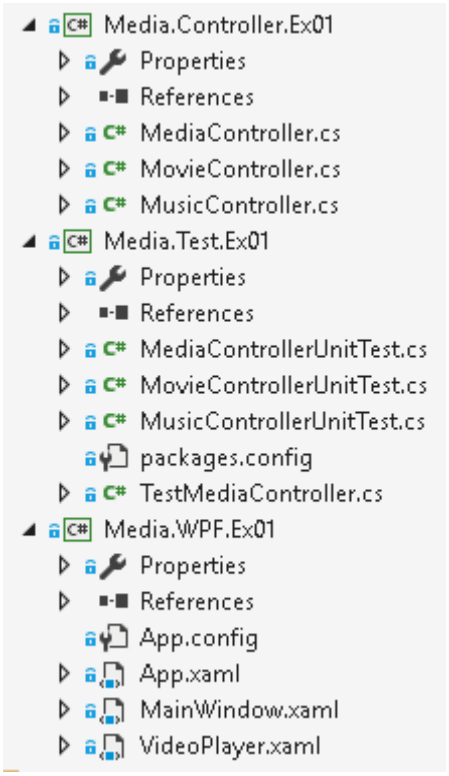
Overview Exercise Programming Advanced .NET

Intro

You will create a WPF application to manage music and movies. To help you getting started you received a project containing the basic structure and a working user interface. Each of the following exercises will expand the abilities of the application and will cover a piece of the course material.

Per exercise there is a solution available. So, if you get stuck or want to see where the exercise is heading to, you can take a peek at the solution.



The start-up project contains the following components:

- Media.Controller.Ex01
  This project contains the controllers. The controllers will do the heavy lifting to keep the WPF layer as this as possible.

- Media.Test.Ex01
  This project contains the tests for the controllers. More on this in exercise 4.

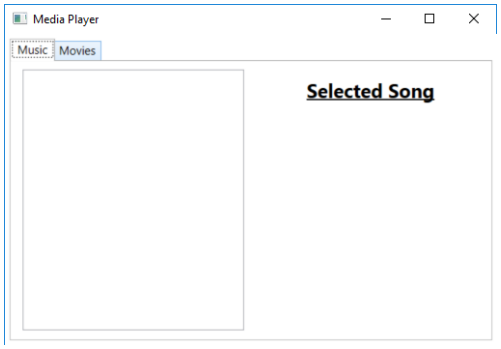
- Media.WPF.Ex01
  This project contains the UI. Try to put only UI logic in this project.

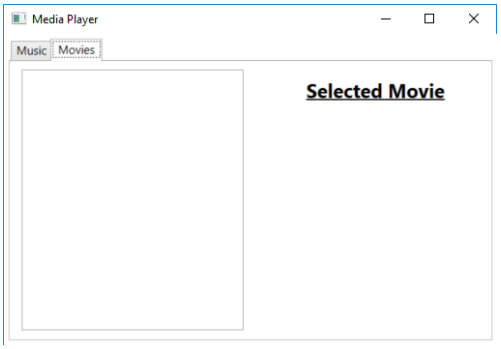
The application will build on the new knowledge gathered during the lessons. You will notice there is only a database added to the application when this part was explained in the course. In the beginning the application will store everything in memory.

Each exercise builds on the previous. So feel free to customize the application to fit your needs. If you could not finish the previous exercise, ask for the solution of that exercise so you can continue with the rest.

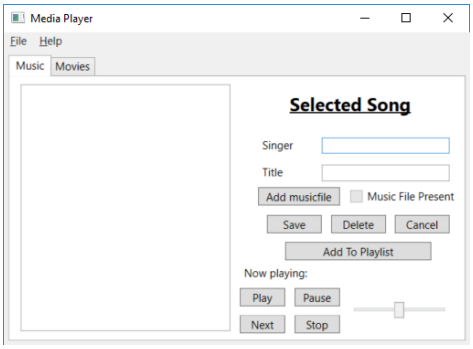
Exercise 1: Controls

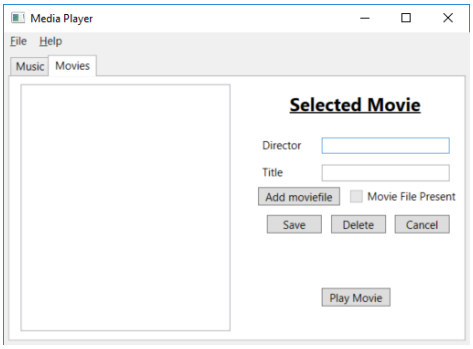
In this exercise we will build the screens of the application. If you run the stat-up project, you will see the screens as shown below.





Add all the controls needed to match the screens as shown below.





Exercise 2: Events

Events enable you to bind actions to controls. The end goal of this exercise is to have a working application. Try not to duplicate code, this helps you to refactor the code as the application evolves and to keep the overview.

Constructor

In the constructor you need to initialize all the components you need lateron. Below you see a code snippet of the constructor. This code is not complete, so, you will need to fill in the gaps.

```
public MainWindow()
        {
            ...

            InitializeComponent();

            _musicController.Player.IsStarted += ...;
            _musicController.Player.IsFinished += ...;
            ...
        }
```

You need

- A controller for the music
- A controller for the movies
- To bind methods to the IsStarted and Is Finished events of the player to change the UI when the state of the Player changes.
- Bind the data of each controller to the correct ListBox

Cleanup

We want our controllers to implement IDisposable. If we do so we need to implement the Dispose method. In this method we dispose all resources used by the controller. For now, only the MusicController needs to dispose the Player. Make sure the dispose methods are called when the windows closes.

TabControl

We use two tabs in the application, one for music and one for movies. To avoid the need to check every time for which tab is active to call the correct controller, you can store the active controller in an instance variable.

When switching the tabs, you also want to set the correct state of the UI.

- The ListBoxes containing the tracks/movies need to be updated.
- Selections need to be cleared
- Set the correct state for the buttons. When nothing is in the playlist, it makes no sense to be able to press the play button.

```
private void TabControl_SelectionChanged(object sender, SelectionChangedEventArgs e)
        {
            if (musicTabItem.IsSelected)
            {
                _activeController = _musicController;
                ...
            }
            else if (moviesTabItem.IsSelected)
            {
                _activeController = _movieController;
                ...
            }
            ...
        }
```

Tip: you might want to look at inheritance for the _activeController field.

ListBox

You want to be able to edit the selected item in the Listbox. In order to do so, the TextBoxes need to be updated. Also the state of the buttons needs to be changed. This can thus be split in two methods.

- SelectMusicItem will update the textboxes and tell the controller which item is selected
- SetMusicForm will handle the buttons.

```
private void MusicListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
        {
            var song = (Song)musicListBox.SelectedItem;
            if (song != null)
            {
                SelectMusicItem(song);
                SetMusicForm();
            }
            e.Handled = true;
        }
```

AddFile

As you might have seen before, there is an “add file” button to allows you to add a file to a record. When a file is available for a record, you will be able to play the song/movie.

This action is the same for songs and movies, so we will implement it only once. There are a few things that need to happen.

- Open a dialog to ask for a file
- Make sure only 1 file can be selected
- Make sure only the correct media can be selected (you might want to have a look at the controllers)
- Use the LoadConvert from the Util project to convert the file into a byte array.
- Place the byte array in a field so you can save it later to the selected record

```
private void AddFileButton_Click(object sender, RoutedEventArgs e)
        {
            var dialog = new OpenFileDialog();
            dialog.Multiselect = ...;
            dialog.Filter = ...;
            ...
            if (dialog.FileName != "")
            {
                _newFile = LoadConvert.ImportFile(dialog.FileName);
            }
            SetMusicForm();
        }
```

Save

Now you should be able to change all the data. To persist these changes you will need to click the save button. When the active controller has no item selected, we add the item to the list.

- Verify that all fields are filled in. The file field is not mandatory
- After the save action is completed, you will need to clear the input fields.
- Update the ListBox as well to see the new record/changes.
- Do the same for movies

```
private void MusicSaveButton_Click(object sender, RoutedEventArgs e)
        {
            if (...)
            {
                if (...)
                {
                    var newSong = new Song()
                    {
                        Singer = musicSingerTextBox.Text,
                        Title = musicTitleTextBox.Text,
                        File = _newFile
                    };
                    _activeController.AddMedia(newSong);
                }
                else
                {
                    var selectedSong = (Song)_activeController.Selected;
                    if (_newFile == null)
                    {
                        _newFile = selectedSong.File;
                    }
                    ...
                }
                ...
            }
            else
            {
                MessageBox.Show("Please fill in all the fields");
            }
        }
```

Cancel

This action will clear all the input fields. This has been done before, so we can recycle the method. In this case we called the method ClearSelection.

```
private void CancelButton_Click(object sender, RoutedEventArgs e)
        {
            ClearSelection();
        }
```

Delete

While adding items is fun, you might want to remove something as well. When you delete a selected item, make sure to remove it from the playlist as well. Afterwards, update the correct ListBox and remove the selection. Removing the selection does not depend on which controller is active.

```
private void DeleteButton_Click(object sender, RoutedEventArgs e)
        {
            ...
            if (_activeController.GetType() == typeof(MusicController))
            {
               ...
            }
            else
            {
                ...
```

```
            }
            ...
        }
```

Playlist

We need to implement an action to add songs to the playlist as well. The MusicController will do most of the heavy lifting. When the song is added to the playlist, we need to update the state of the playlist buttons as well. The SetButtons method will handle the button state.

```
private void MusicAddToPlaylistButton_Click(object sender, RoutedEventArgs e)
        {
            var selectedSong = _musicController.AddSelectedToPlaylist();
            SetButtons();
        }
```

Play

When the playlist contains at least one song, the play button will be active. Make sure the song plays when the button is pressed and the artist and title are visible in the “Now playing” label.

```
private void MusicPlayButton_Click(object sender, RoutedEventArgs e)
        {
            if (audioPlaylist.Count > 0)
            {
                var isPlaying = audioPlaylist.PlaySong();
                ...
            }
        }
```

Navigation buttons

For all the other navigation buttons you can do similar implementations as for the play button. Make sure not to forget updating the state of the other buttons.

Play movie

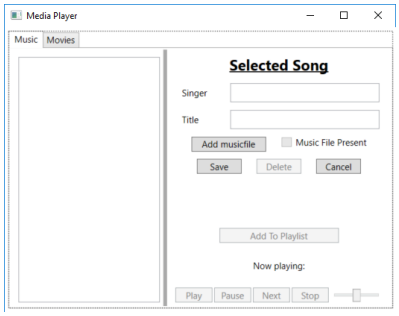
This button will launch the VideoPlayer screen. Make sure to pass the correct data to the video player.

Exercise 3: Layout

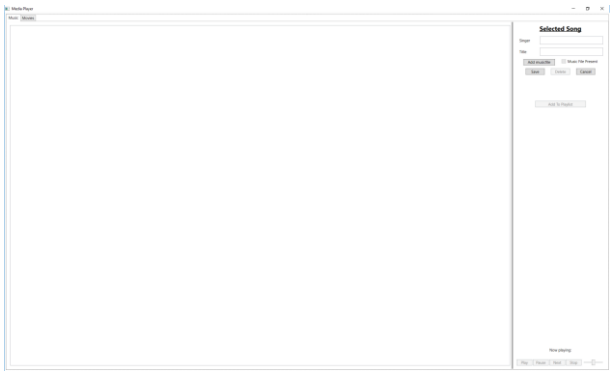
Responsive Design

Until now, when you resized your screen, the position and scale of all the controls never changed. This looks bad and is not usable at all. To make the UI more responsive you will need to use Grid, StackPanel and GridSplitter components. Try to mimic the behaviour as shown below.

Small screen:



Big screen:



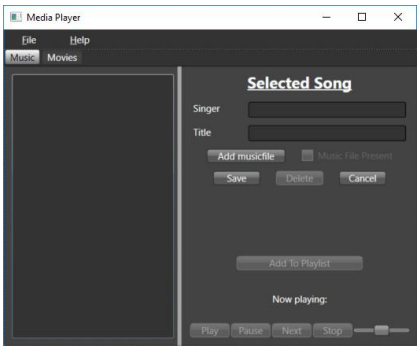
Gridsplitter:

Template

To get a better look on our application, we will use a template. You can find templates in the NuGet repository. We will be using ExpressionDark, but feel free to pick a different one. After installation the template should be added to the App.xaml.

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="Themes\ExpressionDark.xaml" />
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

The application will now look like:

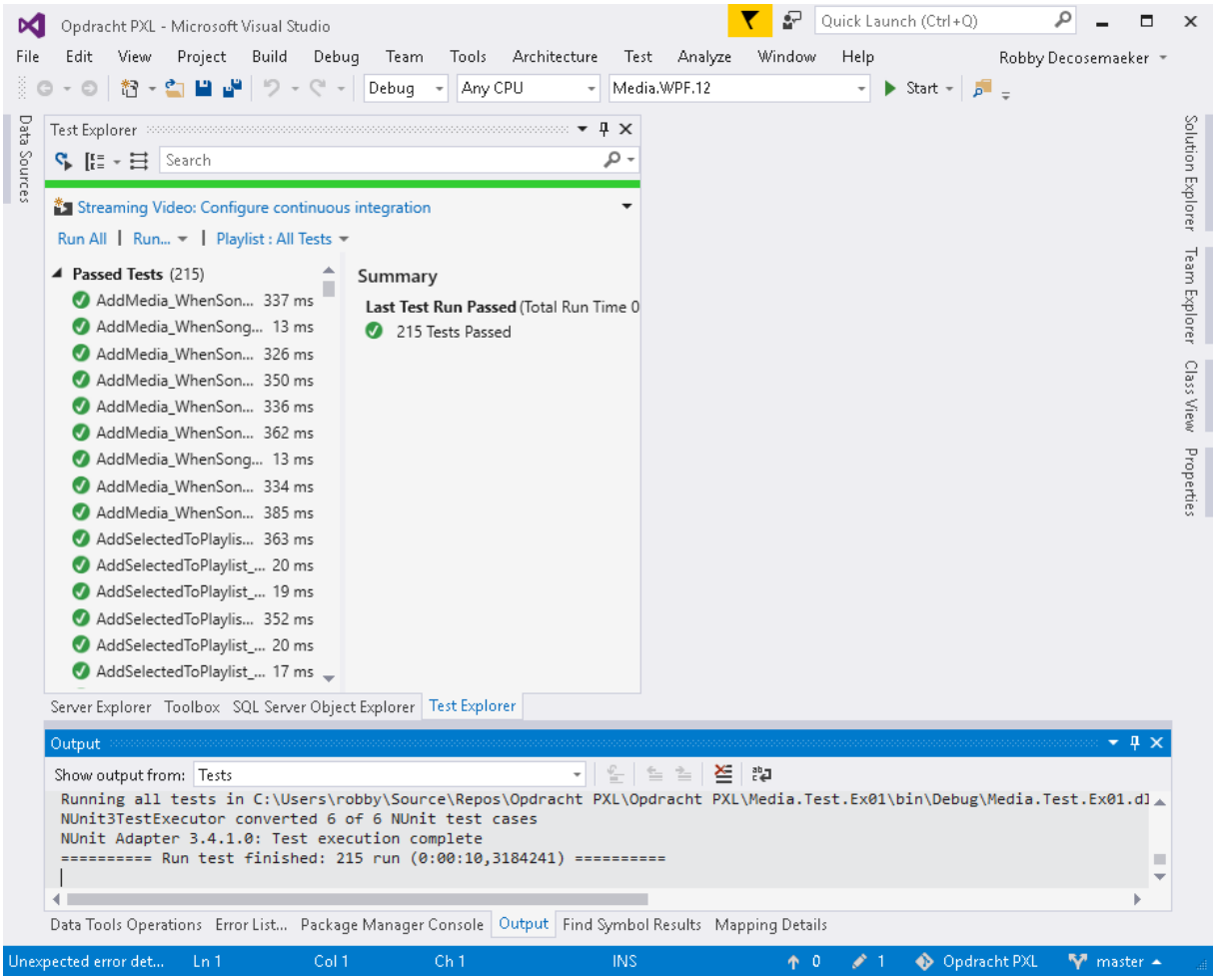


Add a style to give all the buttons the same margin. Do not change or override the template. The only change needed for this will be located in the MainWindow.xaml.

Tip: Base your Style on one of the styles of the Template.

Exercise 4: Unit Tests

Now we have a fully functional application we want to avoid that it breaks when we continue changing the implementation details. A good way to keep track of the functional capabilities of your application is by testing it automatically. Automated testing can be done via unit tests. A lot of testing frameworks are available, but we will use NUnit. You will find a test project in the solution containing already some tests. Install NUnit for Visual Studio and run the tests to verify your installation.



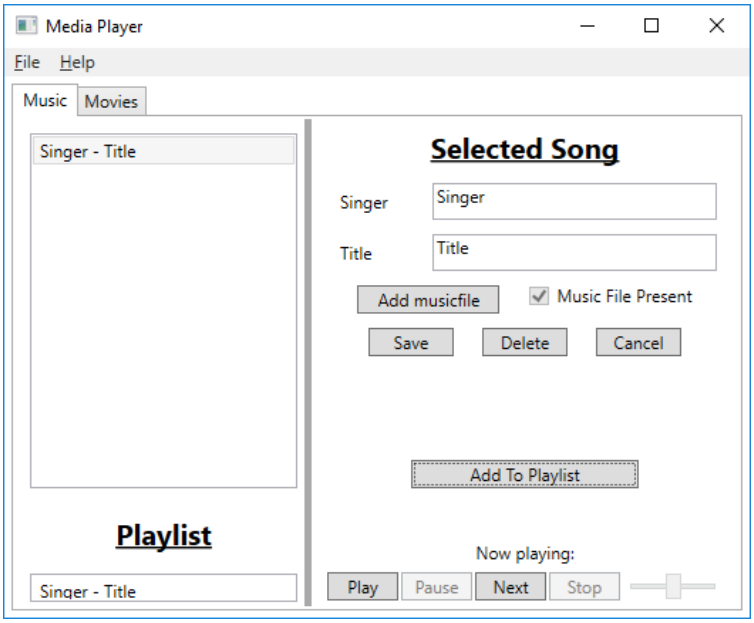
We will only test the controllers. UI testing, Integration testing and System testing will not be covered now. Unit tests will only be used to test logic, not behavior. That is the reason why we only test the controllers. Try to test every method of the controllers. Some methods use external systems, those systems will need to be mocked away. Mocking basically replaces the real implementation with an implementation we control. In that way our tests for the controllers will become independent from the implementation details of the classes they depend on. Mocking will be done using moq. Examples of how to use moq can be found in the test project.

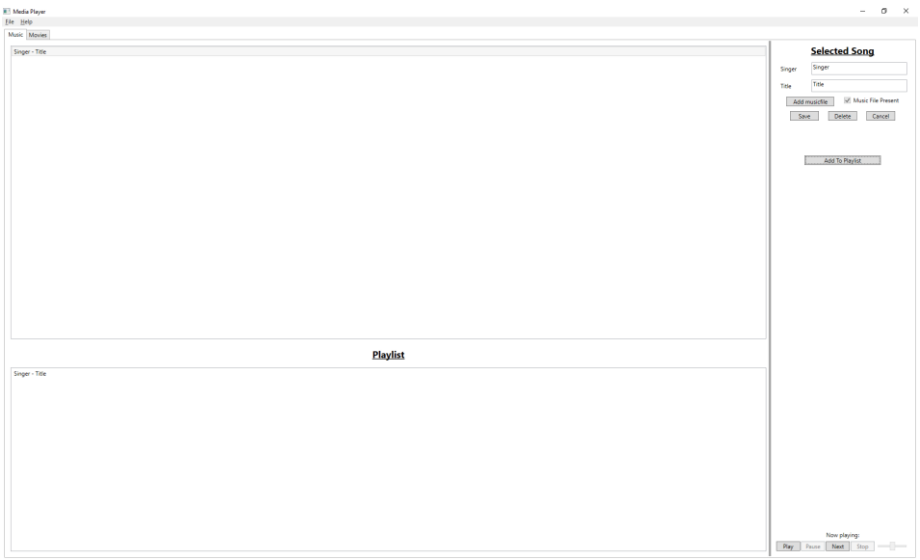
```
38          Assert.IsFalse(_sut.IsPlaying);
39          Assert.IsFalse(_sut.HasSongsInPlaylist);
40      }
41
42      [Test]
        | 0 references | Jonas Annaert, 75 days ago | 1 author, 1 change
43      public void IsPlaying_WhenPlayerIsPlaying_ShouldBeTrue()
44      {
45          //Arrange
46          _mockPlayer.Setup(p => p.IsPlaying).Returns(true);
47
48          //Act
49          var result = _sut.IsPlaying;
50
51          //Assert
52          Assert.IsTrue(result);
53          _mockPlayer.Verify(x => x.IsPlaying, Times.Once);
54      }
```

To be able to test the MediaController separately, there is a simple implementation available in the test project called “TestMediaController”. Use this implementation as a system under test.

Exercise 5: Databinding

Now we will show the playlist in a ListBox via databinding. This enables you to see what is in the playlist at any given time. Add the playlist ListBox to the screens as indicated below.





Keep in mind that the window can resize and you want the playlist to behaves correct in that situation. When a song is selected in the playlist you must be able to edit the fields as you can when selecting a song in the list of all songs. When a song starts playing, it should be removed from the playlist and the title and artist should be available in the “now playing” label.

Exercise 6: ADO.NET

Start by executing MediaDB.sql to create all the tables and stored procedures. After a successful run you can proceed with the exercise.

Until now, rebooting the application resulted in a loss of all changes. To prevent this we will store the movies and songs in a database. To do so we will use ADO.NET to implement all CRUD operations on the database.

To separate the database communication from the UI you will need to put all database related code in a mapper. We call it a mapper, since this terminology is also widely used, but actually we are implementing the repository pattern here. Later on, you will notice this mapper/repository gives you the advantage to be able to change the communication with the database without impacting the rest of the application.

You will need a MusicMapper and a MovieMapper to communicate with the respective database tables. First of all we will define a contract for both mappers.

IMapper

Make sure to keep the contract usable for both controllers, so use Media objects instead of Song and Movie objects. All mapper wil need to be able to perform the following actions:

- GetAllMedia: All data from all records without the ByteArray.
- GetMediaFile: Find the ByteArray of a specific Media object.
- AddMedia: Add a record.
- UpdateMedia: Update a record.
- DeleteMedia: Delete a record.

MusicMapper

First of all we need a way to connect to the database. We will do this by storing the location of the database in a field.

```
public class MusicMapper : IMapper
{
	private string connectionstring = "Data Source=(localdb)\\MSSQLLocalDB;Initial
	Catalog=MediaDB;Integrated Security=true";
}
```

The next step is to get all the Songs from the database.

Tip:
To insert a ByteArray to the database you can use the following snippet, replace [byte array with your own variable]:

```
	"0x" + BitConverter.ToString([byte array]).Replace("-","")
```

Important: This way of storing ByteArrays is not recommended. A better way would be to use stored procedures, like we will see in exercise 7.

The next step is to implement the GetAllMedia method, to retrieve all music records. We only need the id, title and singer for each song. By doing so, we avoid loading all the songs' binary data into memory, we will only retrieve the binary data when we start playing the song. Complete the code listing below, keep in mind to create new exceptions in the datamodel project.

```
public List<DataModel.Media> GetAllMedia()
 {
     var allMusic = new List<DataModel.Media>();
     var query = "SELECT [Id], [Title], [Singer] FROM [dbo].[Song]";

     try
     {
         using (var conn = new SqlConnection(connectionstring))
         {
             using (var cmd = new SqlCommand(query, conn))
             {
                 ...
             }
         }
     }
     catch (SqlException)
     {
         throw new MediaReadFailedException();
     }
     return allMusic;
 }
```

Now we will implement a separate method to load the binary data from the database. We use the id of the song to determine which record we will need. Use the code below as a guide.

```
public byte[] GetMediaFile(int id)
{
    byte[] musicFile = null;
    var query = ...;

    try
    {
        using (var conn = new SqlConnection(connectionstring))
        {
            using (var cmd = new SqlCommand(query, conn))
            {
                ...
            }
        }
    }
    catch (SqlException)
    {
        throw new LoadMediaFileException();
    }
    return musicFile;
}
```

Now we are able to retrieve all the music records, but our database will be empty. To put data in the database you will need to implement a dedicated method. Make sure not to delete the binary data when it is not supplied. When we only save the meta data, you need to check the "File" property to avoid nullpointer exceptions. When the record is saved in the database you can return the id the database generated for the new record.

```
public DataModel.Media AddMedia(DataModel.Media newMedia)
```

```
        {
            string query;
            if (newMedia.File == null)
            {
                query = "INSERT INTO [dbo].[Song] ([Title], [Singer]) VALUES ('" +
newMedia.Title + "', '" + ((Song)newMedia).Singer + "'); SELECT CAST(scope_identity()
AS int);";
            }
            else
            {
                query = ...;
            }
            try
            {
                using (var conn = new SqlConnection(connectionstring))
                {
                    using (var cmd = new SqlCommand(query, conn))
                    {
                        conn.Open();

                        newMedia.Id = (int)cmd.ExecuteScalar();
                    }
                }
            }
            catch (SqlException)
            {
                throw new SaveMediaFailedException();
            }
            return newMedia;
        }
```

The update method looks very similar to the add method. To be able to validate if the update was successful, we return a Boolean indicating if the affected rows is larger than zero.

```
        public bool UpdateMedia(DataModel.Media updateMedia)
        {
            int updateCount = 0;

            string updateQuery;
            if (updateMedia.File == null)
            {
                updateQuery = "UPDATE [dbo].[Song] SET [Title] = '" +
updateMedia.Title + "', [Singer] = '" + ((Song)updateMedia).Singer + "' WHERE [Id] = "
+ updateMedia.Id;
            }
            else
            {
                ...
            }

            try
            {
                using (var conn = new SqlConnection(connectionstring))
                {
                    ...
                }
            }
            catch (SqlException)
            {
                throw new UpdateMediaFailedException();
            }
```

```
        return updateCount > 0;
    }
```

For the delete method you will return a Boolean as well.

```
    public bool DeleteMedia(DataModel.Media oldMedia)
    {
        int updateCount = 0;
        string deleteQuery = "DELETE FROM [dbo].[Song] WHERE Id = " + oldMedia.Id;

        try
        {
            ...
        }
        catch (SqlException)
        {
            throw new RemoveMediaFailedException();
        }
        return updateCount > 0;
    }
```

MovieMapper

The MovieMapper is constructed in a similar way as the MusicMapper. The same actions will be implemented but they act on a different table.

Controllers

To use the mappers in the controllers, you need to add a field of the type IMapper to those controllers. Once the field is instanciated in the constructor, you can start replacing the code that changes the in-memory list to use the new mapper to communicate to the database.

Exercise 7: Parameters

To avoid falling victim to sql-injection attacks, we can use prepared statements instead of plain text queries. To see how a sql-injection attack can be performed you can put ‘;truncate table dbo.Song;-- in the Singer input field and save the changes. This will delete all the songs from the database. After switching to prepared statements, this is no longer possible and the raw string will be put in the database.

Exercise 8: Stored Procedures

Now we have database queries in our code. This is not really convenient, if the database structure changes slightly, the application needs to be recompiled and redistributed. To avoid this, we will switch to stored procedures. By using stored procedures, we can define a simple contract and keep the queries in the database instead of in the application. The stored procedures are already present in the database, you only need to integrate them in the mappers.

By keeping the interface for the mappers, only the code inside the mappers will be affected by this change. This way of working keeps your code easier to maintain.

Exercise 9: Transactions

Before changing a song or a movie in the database, you might want to check if the user is confident to do the permanent change. One way of implementing this is by using transactions. If the user cancels the database operation, you can rollback the transaction.

Add a messagebox in every method to enable the user to confirm (commit) or cancel (rollback) the database operation. For the messagebox to work you will need to link the Presentationframework to the mapper project.

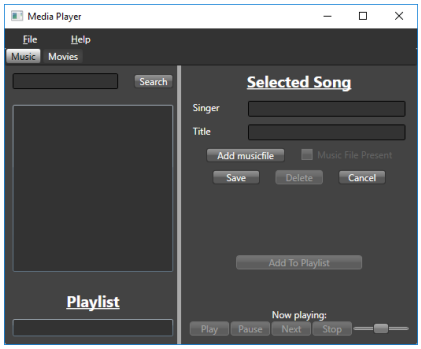
Exercise 10: LINQ

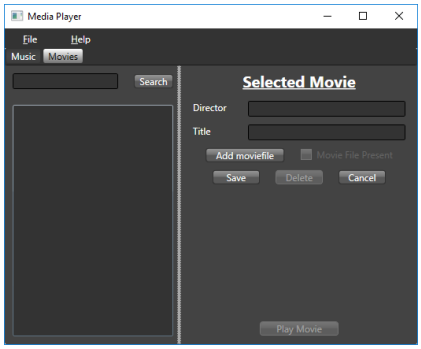
In this exercise you will sort the songs and movies in the list and add the ability to search for a song or movie. The linq queries can be written in the comprehension syntax and in the method syntax. For the application we are writing, there is no real difference apart from the syntax.

First of all, you will sort the songs and music. The sorting will be done in the controller instead of in the database. To implement this, do this in a test driven fashion. First write a test to verify if the songs/movies are sorted. This test should fail at first. Afterwards implement the ordering and see the test(s) pass.

Also searching will happen on the list we retrieved earlier from the database. Therefore we don't need to change the queries to the database. You will need the following components to achieve the search functionality.

- Search(string criteria) method
- A change of the UI





Tip: you might want to look at the inheritance structure of the controllers to avoid repeating yourself.

Exercise 11: Entity Framework

Change the mapper so they use the context of Entity Framework instead of using the ADO.NET classes. You also need to clean the exception handling.

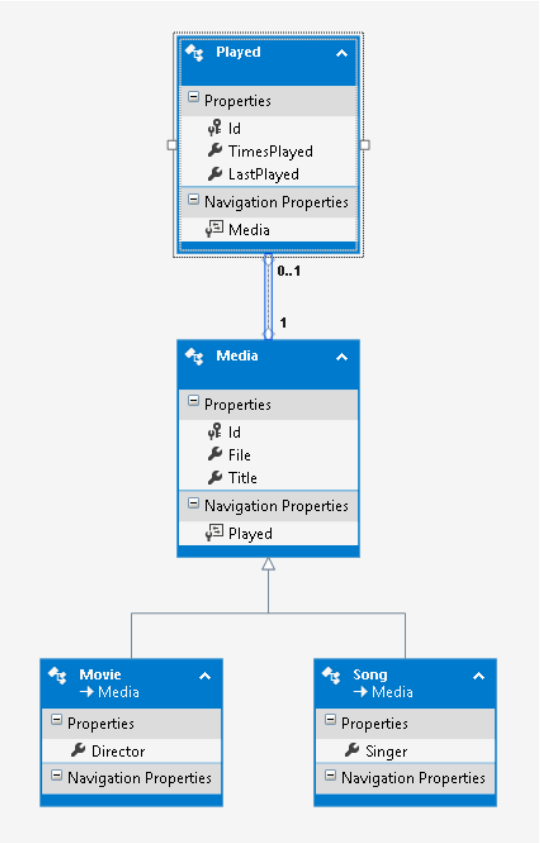
Change the filtering and sorting so it will be done on the database instead of the controllers. If you want to be able to see the difference, try adding logging to the DbContext. For large data sets, filtering and sorting in SQL is faster.

Try to use DataAnnotations as well as the FluentApi to create your datamodel.

Exercise 12: Entity Framework

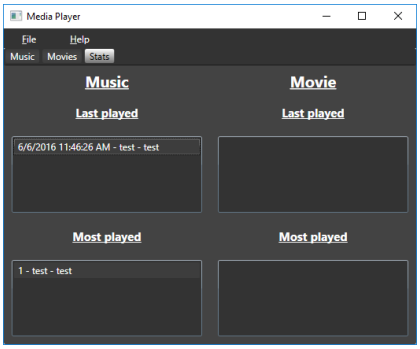
In this last exercise, you will keep track of some statistics. You can see which song/movie was played recently and which are your favorites.

To keep track of these statistics, you will need to add the data to the datamodel. Create a class called Played with the necessary properties. Link the played class to the Media class. When you have done these modifications, your datamodel should look like the diagram below.



Make sure you enable database migrations before changing the datamodel so you can easily modify the database behind. A more drastic approach would be to delete the database and recreate it again.

Now we have all the data, you can start building the screens. Keep in mind the screen can be resized.



To collect the data, we will add a few methods to our MediaController. Make sure to call the AddPlayed method every time you play a song or movie. Also the methods on the mapper need to be implemented.

```
public List<DataModel._12.Media> GetTop10Played()
{
    return _mediaMapper.GetTopPlayed(10);
}

public List<DataModel._12.Media> Get10MostRecentPlayed()
{
    return _mediaMapper.GetMostRecentPlayed(10);
}

public void AddPlayed(DataModel._12.Media media)
```

```
        {
            _mediaMapper.AddPlayed(media);
        }
```

Congratulations! You made it till the end. Please feel free to extend the application with your personal need.

Happy Coding!