**FNDev2 Coursework**

THMusic: Music Collection

**System Design**

**Table of Contents**

[Application Setup 3](#_Toc356144336)

[Solution 3](#_Toc356144337)

[Structure 3](#_Toc356144338)

[Documentation 4](#_Toc356144339)

[Installation 4](#_Toc356144340)

[Coursework Requirements 4](#_Toc356144341)

[Project Requirements 4](#_Toc356144342)

[Functional Requirements 4](#_Toc356144343)

[Additional Features 5](#_Toc356144344)

[Application Components 6](#_Toc356144345)

[THMusic - UI Layer 6](#_Toc356144346)

[IoC (Inversion of Control) 8](#_Toc356144347)

[Core 8](#_Toc356144348)

[Infrastructure 10](#_Toc356144349)

[Design Considerations & Decisions 12](#_Toc356144350)

[Data Model Design 12](#_Toc356144351)

[Asynchronous programming and Concurrency 14](#_Toc356144352)

[UI Design Considerations 17](#_Toc356144353)

[Code Design Considerations 18](#_Toc356144354)

[General Design Principles 19](#_Toc356144355)

[Development Issues 21](#_Toc356144356)

[Async related problems 21](#_Toc356144357)

[Binding to and displaying images from the User (Known) folders. 21](#_Toc356144358)

[Design Time Data in Windows 8 Store (Xaml) 22](#_Toc356144359)

[Problems with Playlist only including certain tracks 22](#_Toc356144360)

[Use of TagLib# and Windows 8 Store app 22](#_Toc356144361)

[Unit Testing 22](#_Toc356144362)

[Evaluation 23](#_Toc356144363)

[Further Development 24](#_Toc356144364)

[Code Refactoring 24](#_Toc356144365)

[Known Issues and Bugs 24](#_Toc356144366)

[Future Development 24](#_Toc356144367)

[References 26](#_Toc356144368)

# Application Setup

## Solution

The project solution is a VS2012 Windows Store Application. Use is made of the MVVMLight framework that gives access to the SimpleIoC: a DI container that works with Windows Store Apps. Included in this framework is a base class that acts as a ViewModel and facilitates implementation of the MVVM pattern. The Taglib# library is also used to interpret the metadata held in audio files to facilitate the inclusion of audio files in the application.

The data is held in an XML file containing various albums in the Music Collection. The contents of this file are loaded at the application start-up to the In-Memory context. All data access within the application is performed against this model. On closure or suspension of the application, the updated contents of the model are again persisted to this XML file. The file is held in the user’s local storage folder.

The solution consists of three pages providing the required functionality

***Main page***: this is the Artist summary page, which shows the albums contained in the collection summarised by Artist. It also displays the summaries by Genre and Playlist.

***Album Details page***: this is a split page, showing the albums contained in the selected group on the left and the details of the album on the right

***Search LastFM page***: this allows the LastFM service to be searched for albums and provides the ability to import them.

***Search MP3 page***: this allows the local machine to be searched for audio files to import into the collection.

The main page has application toolbars that provide access to the categories for grouping the albums as well as access to the Import facilities for LastFM and MP3 and audio files.

## Structure

The project structure follows the [Onion Architecture discussed by JeffreyPalermo](http://jeffreypalermo.com/blog/the-onion-architecture-part-1/), specifically there are separate projects for each component layer.

Core: this is the application core, which includes the domain model and controls access to the infrastructure layer, where the implementation of the interfaces and services are held.

Infrastructure: contains the application cross cutting concerns: the In-Memory context, the repositories and services used by the application.

IoC: dependency injection layer, used to inject the infrastructure where necessary

THMusic: this is the UI presentation layer and contains only UI logic.

Additional folders and projects are

Documentation: this is a solution folder that contains the documentation supporting the solution. It includes this document amongst others.

UnitTests: it was intended that this project contained the unit tests for the various modules of the application, however time scales did not permit any test to actually be written.

## Documentation

The documentation supporting this application are:

SystemDesign.docx: This document

UserGuide.docx: A guide to the application functionality

Readme.docx A quick “getting started” document

THMusic.chm Compiled SandCastle Class Documentation

## Installation

The project can be downloaded from the GitHub repository and should be copied into the projects folder of the VS2012 installation, which must be running under Windows 8. If any Nuget references do not copy, then reapply the Nuget as follows:

MVVMLight: install in the THMusic UI project.

Taglib#: install in the THMusic and Infrastructure projects.

Rebuilding the solution should enable it to run OK.

### Compilation Errors

Please note there will be warning messages produced by the compilation process. These are due to the asynchronous methods not containing any await statements. See *Asynchronous Coding* section later in the document.

# Coursework Requirements

This section lists the requirements from the coursework and how they have been addressed within this solution.

## Project Requirements

This is a Windows 8 Store application, using the .Net Framework 4.5, which is implemented as a Visual Studio 2012 application.

The threading requirements is addressed by the solution using the Task based threading model throughout. It also makes use of monitor lock to control concurrency during updates of the music collection.

Code documentation makes use of the XML style comments that are used by Sandcastle to generate the MSDN style documentation for the modules.

Additional libraries used were MVVMLight and Taglib#, which are loaded to the project through NuGet package manager in VS.

## Functional Requirements

The functional requirements are addressed by the various components of the solutions: specifically:

Type of application: this application is treated like “My Music Collection” which contains references to audio media that I own.

***Browse Collection***: this is provided by the main page, which displays album information summarised by various categories, Artist, Genre and Playlist. Selecting a Group will display the group contents. Eg. Selecting an Artist will display the albums belonging to that artist. Changing the grouping of the collection is accessed through the Top AppBar on the main page.

***Album Details***: the details page, which shows the albums contained within the group, allows for a specific album to be selected, which shows the details, like tracks and wiki information, for the album

***Playing a Track***: this functionality is provided on the Album Details page. If the track contains a reference to a locally stored audio file, the ability to play that has been provided.

***Add to Collection***: this is supported by two features, namely importing information from locally stored audio files, and importing information from the LastFM web service. Both these features are accessed through the Bottom AppBar from the main page.

## Additional Features

The following list describes some of the items included which could be considered as additional features:

* Use of the Taglib# library to retrieve audio file metadata, used to populate the collection album information.
* Globalisation has been designed into the application. Cultures supported should be English, French and German.
* The application is highly structured, based upon the Onion model, including all programming being done to interfaces or abstract classes.
* The MVVM, Unit of Work, Abstract Factory design patterns are employed throughout the application which uses DI to inject the instances of the classes.
* XML Serialisation is used during the persistence of the model data.
* Within the UI, custom controls are used.

# Application Components

Following the structure of the application, described before, the following sections discuss the contents of the components in detail. It describes the function of each component and where if fits into the application structure.

## THMusic - UI Layer

The UI layer is a **Windows Store App** that has multiple pages. It has the main page which shows the current collection contents, grouped by either Artist, Genre or by a Playlist. It additionally provides the ability to search the local machine for audio files and import their information into the collection.

Additional pages show the albums within each grouping alongside the album tracks and other information. There is a page to search the LastFM web service, by entering an Artist and Album title. The option to import this information into the collection is provided from here.

The UI layer makes use of the MVVMLight framework which implements the MVVM pattern as well as provide an IoC container: SimpleIoC.

The three components of the MVVM pattern are provided within the UI project by the following:

* The ***View*** is represented by the application pages. There are three pages, namely
  + ***ItemsPage.xaml*** : The main Albums Summary page showing the albums summarised by the possible groupings, Artists, Genres or Playlists.
  + ***SplitPage.xaml***: The Album Details page, which shows the albums included within the group and the details of each album, when selected.
  + ***ImportLastFM.xaml***: The LastFM Import page, which supports the search facility for LastFM.
* There is a ***View-Model*** to support each page in the application. They can be found within the ***ViewModel folder*** of the project.
  + ***MainViewModel***: this view supports the Album Summary page, consuming its data-models and data services required to populate them.
  + ***AlbumsViewModel***: this supports the Album Details pages, consuming its data-models and data services required to populate them.
  + ***LastFMViewModel***: this supports the LastFM Import page, consuming the LastFM data service.
* The ***Model*** for each view is represented by two separate components:
  + ***Data-Models***: these are UI representations of the data required to support each view. They are located within the ***DataModel folder***. Each model implements the ObservableObject base class provided by the MVVMLight framework. This has the expected notification of properties changing required for the UI binding. They are loaded through the ViewModel for each page.
    - ***AlbumModel***: this contains the details for each album displayed in the UI.
    - ***GenreModel***: this contains the genre related information for each album. It is reference in the AlbumModel.
    - ***GroupModel***: this model show the summary information supporting the main page, the summaries are Artist, Genres and Playlists.
    - ***GroupTypeEnum***: describes the type of grouping available for the album information.
    - ***TrackModel***: this contains the information for each individual track with an album.
    - ***WikiModel***: this contains the LastFM wiki information for an album.
  + ***Data-Services***: These control access to the other layers of the application and are called from within the ViewModel for a page. They gather the information for the data-models and the views within the application. These data-services can be found in the ***Data folder***, which includes the interface describing the contract for each service and allowing it to be injected into the ViewModel at runtime. The data-services provided are:
    - ***AlbumModelDataService***: controls access to the core project for the all information related to the albums. It supports the ***AlbumsViewModel***. Its interface is ***IAlbumModelDataService***.
    - ***GroupModelDataService***: controls access to the core project for all summary information of albums, for the possible groupings: Artists, Genres and Playlists. It supports the ***MainViewModel*** and its interface is ***IGroupModelDataService***.
    - ***LastFMModelDataService***: controls access to the LastFM service provided by the core project. It supports the LastFMViewModel has interface ***ILastFMModelDataService***.
    - ***MusicFileDataService***: controls access to the MusicFileService provided by the core project which imports the audio files. It is used by the MainViewModel, with the interface IMusicFileDataService.

The other folders in the UI project contain the following components:

* ***Common***: In addition to the standard components in this folder, a ***CustomStyle.xaml*** file has been created to provide styles specific to the application.
* ***CustomControls***: This folder contains the xaml user controls used within the UI.
  + ***AlbumAndTrackPlayer***: this control is used in the Album details page, to allow for the media player element to be included
  + ***AlbumAndTrackView***: this has no media player element, and is used by the ImportLastFM page.
  + ***MusicPlayer***: this control contains the MediaElement and provides the ability to play the track, provided the audio file exists.
  + ***AppBarBotton***: this provides the application bottom appbar, which is accessible from the main page. It provides access to the LastFM search and Import MP3 facilities.
  + ***GroupAppBarTop***: this provides the ability to change the main view groupings. It allows the group summaries to be switched amongst Artists, Genres and Playlists
* ***Design***: This folder contains the design time data sources, and the interfaces that allow them to be injected into the view-models allowing the xaml editor to show realistic controls as they are added. See *Design Time Data* within the *Known Issues and Bugs* section of this document.
  + ***DesignAlbumModelDataservice***: the design time data for the AlbumModel which support the album detail page
  + ***DesignArtistRepository***: the design data for the Main page.
  + ***DesignGroupModelDataService***: assists in the design data for the Main page
  + ***DesignLastFMModelDataService***: This supports the design time data for the ImportLastFM page.
  + ***DesignMusicFileDataService***: this is a stub to ensure that al injected instances are resolved at design time.
* ***Helpers***: This folder contains various helpers used within the UI layer.
  + ***ImageLoader***: this supports the UI directly by resolving the image paths held in the data model to provide the actual bitmap images shown in the UI.
  + ***LocalisationHelper***: this encapsulates the functions required to access the localisation features provided by the application. It provides a simplified api to the resource loaders required for localising things like date and number formats.
* ***Navigation***: This folder contains the navigation service and the interface to allow it to be injected into the view-models. This allows the navigation between pages to be initiated from within the view-model.
* ***Strings***: This contains the resource files required to provide the various cultures the application supports. It should be French and German[[1]](#footnote-1).
* ***ViewModel***: This folder contains an additional class, ***ViewModelLocator***. This is the MVVMLight module that controls the instantiation of the various ViewModel classes, but also sets up the IoC container controlling the injection of the modules at runtime.

## IoC (Inversion of Control)

The SimpleIoC container is used for dependency injection. It comes as part of the MVVMLight framework used for the application. This was the motivation for using this framework as one of the few, if not, the only IoC container at the moment which installs and works within the Windows Store App environment.

It contains a single class, Configuration, which is where the cross-cutting components are registered with the IoC. Other components specific to the UI are configured within the ViewModelLocator class, described above.

## Core

This project contains the Data model for the Music Collection. It contains the Albums, Tracks and associated information for Artists, Genres and Playlist.

It exposes various interfaces to the UI layer that give access to the required data services and others. Additionally it provides the factories for creating domain model classes elsewhere within the application.

The Core project is structured so that each component is logically placed within it, describing its intended use.

### The Model folder

The Domain model classes are included in the Model folder and are as follows:

* ***Album***: describes the album information
* ***Artist***: describes the associated Artist information
* ***Genre***: describes the associated Genre information
* ***Playlist***: describes the associated Playlist information
* ***Group***: is the base class for the Artist, Genre and Playlist classes. This provides the ability to group albums using generic routines, with this class providing the base functionality.
* ***Track***: describes the information for the Tracks belonging to an album
* ***Wiki***: describes the wiki information associated with an album
* ***Image***: references the images associated with an album.
* ***ImageSizeEnum***: defines the various image sizes available.

All the above classes are abstract and there is a subfolder containing the classes that provide concrete implementations of these classes

### The Factories folder

This folder contains the factories used to create instances of the abstract model classes. The factories follow the ***AbstractFactory*** pattern, see *Patterns* in the *Design Considerations & Decisions* section. There is, therefore, an abstract factory that defines the common functionality. There are model class specific factories, some of which also use factories to create related classes.

### The Interfaces folder

This folder contains the interfaces to the various repositories within the Infrastructure, which the UI later requires through its data services. It contains:

* ***IUnitOfWork***: this provides the interface for the In-Memory data context, which contains the physical data for the Music collection. It follows an implementation of the ***UnitOfWork*** pattern, see ***Patterns*** of the ***Design Considerations & Decisions*** section.
* ***IAlbumRepository***: provides the interface to the AlbumRepository that access to the Album related information.
* ***IArtistRepository***: provides the interface to the ArtistRepository that gives access to the Artist related information.
* ***IGenreRepository***: provides the interface to the GenreRepository that gives access to the Genre related information.
* ***IPlaylistRepository***: provides the interface to the PlaylistRepository that gives access to the Playlist related information.
* ***IGroupRepository***: serves as the base interface for the IArtistRepository, IGenreRepository and IPlaylistRepository. Again it supports the use of these repositories in generic routines through polymorphism.
* ***IPersistencRepository***: this is a special repository which provides the functionality to load the model from and persist the mode to the underlying XML file. It is provided as a separate repository so that the functionality can be called from the App.Xaml module. This is specific to Windows 8 Store applications.

### The Services Folder

This folder contains contracts describing the functionality provided by the services consumed buy the UI. It contains the interfaces and the supporting data model. These data models are held here as they are not part of the domain model, merely transient DTO’s.

* ***ILastFMService***: the contract for the LastFMService
* ***LastFMAlbumInfo***: contains the information retrieved from the LastFM album.getinfo service
* ***IMusicFileService***: the contract for the MusicFileService
* ***MusicFileInfo***: contains the information retrieved from the audio files, accessed through the MusicFile service.

## Infrastructure

This project provides the services and accesses the data that supports the core project domain model and services. Again it is separated into projects that imply the intent of their contents. Specifically they are:

### The Data Folder

This folder contains the modules that relate specifically to the In-Memory context, which contains the actual data supporting the domain model, see *Data Model Design* in the *Design Considerations & Decisions* section. These modules are:

* ***MusicCollection***: contains the collections holding the album and related data. There a routines, internal to the context, for adding albums and adding tracks to albums.
* ***DataHelper***: this class contains the logic for updating the albums and related collections within the Music Collection.
* ***BuildNavigationProperties***: this class is called when the Music Collection has just been loaded at application startup, and rebuilds the navigation properties of the classes within the domain model.
* ***InitialiseMusicCollection***: this is called once, when the application first starts. The underlying XML file will not exist to there will be no data to load, so this routine supplies some seed data. It will be persisted on the first closure of the application.

The updates to the collections within the Music collection are controlled with manual monitor locks to ensure they are thread safe. Again, see *Data Model Design* in *Design Considerations & Decisions* section.

### The Helpers Folder

This folder contains helper classes’ specific to accessing the underlying file structure. The helpers are:

* ***FileIOHelper***: provides a simplified api to the underlying asynchronous Windows.Storage file access methods.
* ***DataFileHelper***: supports accessing the underlying XML data file.
* ***MusicFileHelper***: supports the accessing of audio files.
* ***SerialiseHelper***: provides the serialisation and deserialization of the underlying XML data file.

### The Repositories Folder

This folder contains the various repositories which provide access to the Music collection. The repositories are:

* ***BaseRepository***: this is an abstract generic base repository supporting all other repositories
* ***AlbumRepository***: controls all access to the Album information
* ***ArtistRepository***: controls all access to the Artist information
* ***GenreRepository***: controls all access to the Genre information
* ***PlaylistRepository***: controls all access to the Playlist information
* ***PersistenceRepository***: controls access to the loading and persisting of the Music collection at application start-up and shutdown.

### The Services Folder

This folder contains the implementation of the services not related to the domain model. They are:

* ***LastFMProxy***: this is a proxy service to the LastFM web service, specifically the album.getinfo service.
* ***LastFMHelper***: this helper class performs the extraction of data from the XML returned from the lastFM service and maps it to the LastFMAlbumInfo class passed back to the application.
* ***MusicFileService***: this uses the Taglib# library to encapsulate access to the metadata for audio files.
* ***MusicFileMapper***: this mapper class support the MusicFileService by mapping the Taglib information to the MusicFileInfo class, passed back to the application.

### The Design Folder

This folder contains what would be a data model used for unit testing. It was originally intended to support the design time data, but that is better served directly within the UI layer. It is currently not used within this application.

# Design Considerations & Decisions

This section covers the main decisions made during the design of the application.

## Data Model Design

The domain model was set up to reflect the natural relationships between an Album and its tracks, the associated Artist and Genres. The addition of playlist into the model, logically associates a playlist with a track. This supports the ability to create a compilation of favourite music for example.

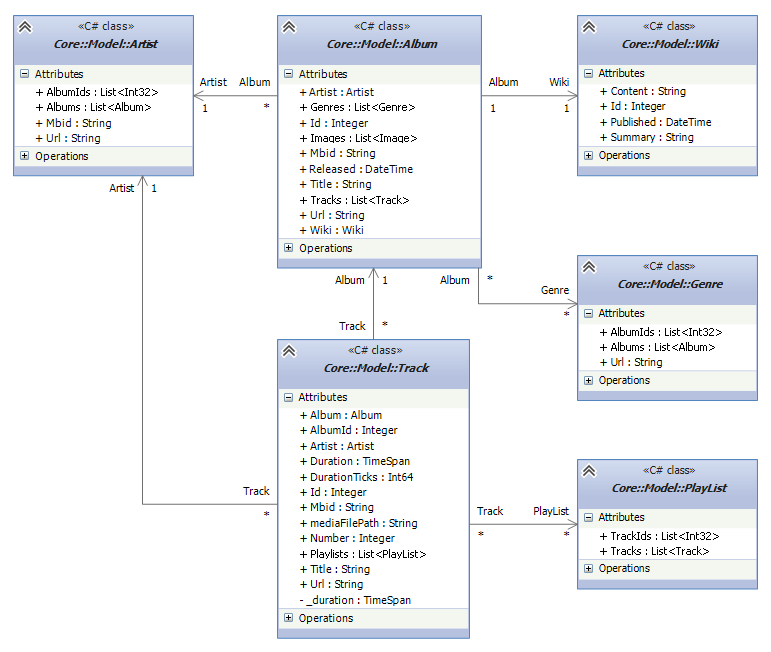


Figure - The Domain Model: Class Diagram

This model was designed along these lines but with the inclusion of navigation properties to support defining ***Linq-to-Objects*** to query without the use of joins. This model is populated quite easily, although there are some problems, not fully appreciated when combined with the decision to persist the data within an XML file, that is loaded at start-up and persisted at shutdown.

The use of SqlLite, was considered, but with no experience of this product whatsoever, it was decided to use the XML approach. Additionally, the amounts of data for this implementation will be extremely small and therefore the serialisation processes will not hold the application up significantly. This would only be apparent at start-up anyway.

This was sound decision, however, the problems associated with it concern,

* The persistence or serialisation of the model
* The updating of Albums within the model

### Serialisation.

The serialisation problems were caused by circular references due to the navigation properties. These had to be removed before the object could be serialised. Possible solutions were

Solution 1: Persistence Model and Mapper classes

This involved setting up a Persistence model within the Data layer substituting the Navigation properties with the Id of the related class. It also required two mapping classes, to map between the domain and the persistence models. The logic of the mapper routines, however, are necessarily complex to be able to reconstruct the navigation properties.

This solution offered the best separation of concerns between the models. The domain model has no knowledge of the persistence layer at all, whereas the persistence model is decorated with the necessary XML attributes to ensure the XML serialisation worked.

Solution 2: Customised XML Serialiser

Another option was to look at customising the ***XMLSerialiser*** in the Windows 8 Store environment. This is not quite as straight forward as first thought, and involves implementing the ***IXmlSerialisable*** interface methods are: ***WriteXML*** and ***ReadXML***. Both these methods take over the entire process: you have to write the complete serialise and deserialise processes. This would be similar to the mapper methods in the first solution but also writing the actual XML itself.

Solution 3: Add XMLSerialise Attributes to the Domain Model

The option finally taken was to include ***XMLAttributes*** in the Domain model itself, avoiding the need for two separate models in such a small application. Additional properties were added to break the cyclic references and the navigation properties they matched were marked with ***XMLIgnore()*** so that they were not persisted. This involved a single custom mapper routine, whose purpose is to build the navigation properties to complete the domain model and populate the other related collections.

This 3rd options was chosen as ***it*** offered the simplest solution. However it introduces persistence concerns into the domain model. With it being such a small application, the reduction in the code required, the simplification of the mapper to rebuild the navigation properties, this was the preferred option.

### The updating of the Album within the model.

Routines were required to ensure the updates to the related collections and navigation properties were kept in a consistent state. These were small routines, designed to check if the object already exists and updating or adding it if the object is new. Once these updates are done the object created is then added back to the album record ensuring the navigation properties are correctly updated.

These routines are abstracted to the DataHelper class to separate the concerns and observe the principle of single responsibility. The updates themselves are part of the Music collection context, as any repositories are not concerned with how the updates are done within the model.

See *Evaluation* section for thoughts on this subject.

## Asynchronous programming and Concurrency

### Asynchronous Coding

Making code run asynchronously is not as simple as it seems, considering the new Task based model introduced by the .Net Framework 4.0. Features available are

* The ***Task*** class and the ***await*** keywords simplifies the coding required, which takes the place of coding the callback methods and delegates.
* There is also support provided by the parallel versions of ***foreach*** for example
* There is PLinq, which supports parallelised queries, not complete support though

The multiple threads mean that concurrency must be considered when performing updates on objects in a thread safe manner. For this there are

* New Concurrent collections, which have the necessary locking mechanism built in.
* The use of the various manual locking mechanisms to ensure thread safety

There are a lot of things to consider, but they are not always appropriate within an application. To use these effectively they have to be included within the design from the start. It is not something that can be retro-fitted, not easily anyway.

The areas of concern can be split as follows

* The UI layer must call objects asynchronously to keep the UI responsive.
* Communication between the various layers of the application must be kept asynchronous so as not to block either
* The updates within the data layer should be kept as asynchronous as possible.
* Additionally, the concurrency of the updates within this layer must be carefully controlled if there are multiple threads possibly performing update. This would involve some kind of locking mechanism to address the thread safety of the objects being updated.

Within this design these concepts were adopted and implemented as follows:

###### The UI Layer

The UI constitutes the ViewModel, from which all calls that retrieve data are made asynchronously, therefore calls to the data services are also made asynchronously: eg. The GetLastFMAlbumInfoAsync method called from within the LastFMViewModel.

Calls from the data services to the back end are also done asynchronously to keep the chain going and keep the UI free. These are calls to the various repositories

The supporting data models, like the AlbumModel for the album detail page, all load images, and the calls to do this are asynchronous too, they all call the ***LoadImageAsync*** method of the ***ImageLoader*** module.

The event handlers, for the button clicks are also handled asynchronously by the viewmodel. Examples are on the LastFM import page, namely the ***LastFMSearchCommand***, which initiates the call to the web service, and the ***LastFMImportCommand*** which updates the results of the search.

Making use of the binding properties of the Xaml based UI, and the use of Observableobjects (the MVVMLight equivalent of Observable Collections), all help free the UI and ensure it’s updated appropriately.

###### Repositories

The methods of the repositories are marked as async so they can be called asynchronously.

The ***get*** methods however access the in-memory context using Linq-to-objects. These are extremely quick and are either passing back the entire collection of albums, artist etc. or they are returning a single element.

While PLinq could be used, there is no scope here to take advantage of. Therefore the use of Plinq is considered unnecessary. Essentially these methods run synchronously, and produces many warnings about the 'lack of await' within the method so they run synchronously anyway. The use of Plinq doesn’t remove the warning message, even if it is working in a parallel fashion.

The AlbumRepository is the only one that has update routines. They are the CreateAsync and the AddTrackToAlbum. These call the update methods of the Music collection asynchronously.

The PersistenceRepository calls the Load and Save methods asynchronously

###### Services

The services are all asynchronous methods as the make calls to the internet or to the underlying file system and such calls are all async in nature. Specifically:

* ***LastFM***, this is most definitely an async task
* ***Image*** loading, this is also an async task, as the path is supplied, and it's the responsibility of the UI to actually load the image.
* ***MP3*** loading using Taglib# is most definitely async as the file IO for a starter is async.
* ***File IO Helpers***: these are all async.

###### Bottlenecks, Synchronous methods

There comes a point where it is no longer possible to keep the methods executing in an asynchronous way. Such examples exist in this application. Eg. the ***LocalisationHelper*** class in the UI, some of the methods in the repositories. The Factories of the Core project are another example of such bottlenecks. This is not a problem as long as they are short tasks to complete, like those mentioned are. See *Compilation Warnings* in *Async related problems* section for associated issues.

### Concurrency of data updates

Within the multi-threaded application the update of data models can be attempted from more than one thread. Such collections therefore have to be thread-safe and support the various locking options available to ensure safe update of items. Even though this is a single user application the async model is used throughout the application to keep the UI responsive, but means that the possibility exists that updates are concurrently performed from different threads.

The principal area of concern is the In-Memory collection that supports the Domain Model and keeps the entire Music collection available. The only sources of update are from the LastFM import and the MP3 import. All other access is simply as the result of Read operations.

Serialisation must be considered carefully as should all file access. The Windows.Storage is designed to only work asynchronously and it can be safely assumed that access to files has therefore been made thread-safe.

Certain lists are used elsewhere in the application, but are created and completely updated within the modules so should not be affected by multiple threads updating them. They are passed between method calls but once this happens they are used to iterate and no updating of them occurs outside the methods that create them so they are safe to leave as List<T> collections.

The .Net Framework provides various options to ensure the thread safety of collections, which are:

1. ConcurrentBag
2. ConcurrentDictionary
3. ConcurrentQueue
4. ConcurrentStack
5. Manual Locks during updates of List<T> collections.

Options 1 to 4: Concurrent Collections, are recent additions to the framework specifically designed to be thread safe by having built-in locking mechanisms. The last is the same locking mechanisms that have been around since version 2.0.

###### In-Memory collections update requirements

The application uses Linq to access the collections directly. They are also iterated over to process them. Updates are performed against specific elements of the collections. Therefore the selection of the correct option, the following requirements must be met.

Compared to the

The Concurrent collections must support the following requirements and functionality at least.

* Must support enumerations over the collection
* Must support the use of Linq-To-Objects to retrieve information
* Must support the ability to delete the item
* Must support the ability to replace/update an item. This could be Remove the item, update it and add it back.

###### Comparisons

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Collection | Enumeration | Linq | Retrieve Specific Item | Update Specific Item | Delete Specific Item |
| ConcurrentBag<T> | Yes | Yes | No | No | No |
| ConcurrentDictionary<T> | Yes | Yes | Yes | Yes | Yes |
| ConcurrentQueue<T> | Yes | Yes | No | No | No |
| ConcurrentQueue<T> | Yes | Yes | No | No | No |
| Manual Locks (List<T>) | Yes | Yes | Yes | Yes | Yes |

 See References section for sources of the research.

The Queue and Stack provide no methods of retrieving specific items from the collection and therefore are not appropriate.

The ConcurrentBag, allows an item to be removed from the bag, but it is not possible to specify which one, it is simply a question of getting an element. This would not support the necessary requirements either.

The ConcurrentDictionary could support retrieval of specific objects, but is not suited to the application structure without significant modification.

The approach adopted was to use a Monitor lock on the List<T> from within the method that updates are performed. Because the update methods have been sufficiently abstracted with separation of concerns in mind as well as the single responsibility principle, the only update routines are in the DataHelper class in the Infrastructure.Data namespace. It is fairly straight forward to incorporate this locking mechanism within these methods.

See *Evaluation* section for some thoughts on this topic.

## UI Design Considerations

### Supporting Navigation between pages, passing parameters and controlling from the ViewModel

 I have followed guidelines from an MSDN magazine article on how to control navigation between the pages (Frames; not actual new pages, but frames within the main App) and to pass parameters to them, in such a way as to allow separate ViewModels, one for each page of the application.

This article, written by Laurent Bunion, a main author of MVVMLight.

<http://msdn.microsoft.com/en-us/magazine/jj651572.aspx>

Basically you provide a Navigation Service, that implements an INavigationService interface, which can be injected using the SimpleIoC. It allows navigation to be triggered from processing within the ViewModel. The parameters that are passed, migrate their way through to the ***LoadState*** method of the target page and it is possible, to pass values into the viewModel at that point, providing access to the instance is obtained through the IoC container.

This is not the most elegant way, or the most decoupled way to pass parameters between forms, but it's as far as I've taken it just now.

Another article, by Geoff Webber-Cross, demonstrates a way of passing parameters in a decoupled way, by making use of the Messaging service provided as part of the MVVMLight Tooklit. It gets a Thumbs-Up from MR Laurent Bunion. Time constraints do not permit implementing this here.

### UI Custom Resources

The design includes the use of custom styles and user controls to control displaying of information on the UI. The decision whether to use a user control or a custom style, was broken down to the following: User Controls where many components are required and possibly the inclusion of custom functionality, like the MusicPlayer User Control. For specific small items, like listing an Album header or Track information within other controls like ListViews, then the Custom style was deemed appropriate.

### Adding Custom Styles added as a Resource Dictionary

The rest of the common layouts, like the ***Group250x250ItemTemplate***, ***Group80Itemtemplate***, ***Album130ItemTemplate*** and ***Track80ItemTemplate*** have been added as ***CustomStyles*** and added as a Resource through the ***Merge.Resources*** in the ***App.Xaml*** file. This is a much more re-usable way of doing things, instead of User controls, it gives the freedom to use within other controls just the same, but is less work to set up. User controls are best used when functionality as well as layout needs to be encapsulated.

### Adding the media player to the App

The MediaPLayer is designed as a user control to specifically encapsulate the functionality of the buttons to play etc. within the control.

Accessing the button events and controlling the source is straight forward in the code-behind the control. However this must be designed to be controlled from the view model. It must be possible to display or hide the control when a track is “playable” or not.

Testing of using the MediaElement did not seem to work by binding the Source to a property of the view model. However, using the SetSource method, which can only be done from code-behind with a stream to the source file, and the player works fine.

It is imperative to avoid breaking the MVVM pattern. This is achieved using a hidden property included within the MediaPlayer user control, and this is bound to the media file property allowing the file to be set from the view model. The visibility is also controlled with an ***IsPlayable*** property, which is converted with the ***BooleanToVisibilityConverted***. The ***IsPlayable*** property returns true if the track contains a reference to a locally stored audio file. The playing is controlled from the buttons included within the user control.

## Code Design Considerations

### Design code structure to support Groups of Artist, Genre or Playlist

The problem considered here was how to use a single, generic, method to allow the repository to be used for each grouping class, without having to use a separate method for each.

The solution was to

* Define an IGroupRepository Interface, which contained the common methods for each of the types.
* The IArtistRepository, IGenreRepository and IPlaylistRepository then derive from this base interface.
* Define an abstract base Group class containing the common properties of the types, namely the Id and the Name.
* Derive the Artist, Genre and Playlist classes from the Group class.

This allows the appropriate repository to be injected at runtime into the GropupModelDataService, but where necessary it can be cast to an instance of IGroupRepository, which will then allow the repository to be used within a generic routine for any of the classes. It also allows the Id and Name properties to be set within the generic method. This is used within the GroupModelDataService.

It was necessary to consider this, as a way of minimising code and adhering to the DRY principle.

Because the linq used within the repositories and the fact that linq queries use different collections, it was not possible to implement the queries in the base repository. This would not be a problem within the Entity framework environment, but here it is an issue because the collections are not easily interchangeable within each query itself.

### Importing MP3 Files: Adding MP3 tracks to the collection or an existing Album

The current functionality designed into the audio file import is as follows:

* It is possible to add a complete Album from LastFM
* It is possible to add a locally stored MP3 to a an existing album, or to create a new album
* It is not possible to add the same album multiple times through LastFM.
* It is not possible to add the LastFM information for an album that was created when adding an MP3.

Other design consideration are:

* Artist Name: the ***Artists*** in the Taglib# file is marked as obsolete. However, if the preferred properties of ***albumArtists*** or ***Performers*** to not provide the information it accesses the obsolete property. This is to support older audio files.
* Where the MP3 file belongs to a new album, then a call top the LastFMSerivce is made to get missing information, specifically the images. If there are none, they are default to image sin the Assets folder.
* When adding multiple MP3 file for a single application, the calls to the LastFM service should be cached to avoid calling the service multiple times for the same information. See *Caching the LastFM calls* section for this as it is not currently implemented, but will make a significant improvement on performance of the Import MP3 functionality

## General Design Principles

### Patterns used

To improve separation of concerns, the ‘onion model’ as previous mentioned, was adopted. This relies on the use of Dependency injection to inject the infrastructure components as required.

Within the UI layer the MVVM architectural pattern is employed, with the MVVMLight framework, which also supplies the IoC for the DI.

The Abstract Factory pattern is used for creation of class instances, within the core project. The UnitOfWork pattern is used to ensure the same instance of the MusicCollection in-memory context is used by all repositories trying to access it.

### Frameworks and API’s

###### MVVMLight

This is a framework that provides an implementation of the MVVM pattern, by supplying a base ViewModel class from which the application classes are derived. It also includes an IoC container called SimpleIoc, which is about the only container for implementing DI I can find that currently appears to work with Windows Store Apps.

Having implemented the MVVM pattern in the FNDev1 coursework, I thought it would give a good comparison to use this framework. I checked with the course tutor to confirm that use of this framework would be acceptable for the coursework. Confirmation was duly received hence its inclusion within this project.

###### Taglib#

This is a library that provides the ability to access the metadata contained within audio files. This metadata contains information about the track, the album, artist and possibly genres. It is used within the MusicFileService of the Infrastructure project.

### Data Storage

Provided data is as a text file, specifically CSV format, therefore file access is using a TextReader. The XML serialiser is used for reading and writing the searches file.

### Design Instance of Data

The design time data facility is provided as part of the MVVMLight framework. The design time data sources are injected using the IoC to provide data for the xaml designer.

Designing this takes care as it is not possible to debug it, and if errors occur they take time to resolve. The basic idea is to keep the provision of this design time data as simple as possible.

# Development Issues

## Async related problems

### Compilation Warnings

When a method that is marked with the ***async*** keyword because the calling method is ***async*** and it contains no ***awaitable*** calls the compiler issues a warning stating this fact. This is often unavoidable due to one of the bottlenecks referred to earlier. There are quite a few instances of this situation within the application and therefore there are correspondingly lots of these warning.

The use of PLinq within the repositories, which is where some of these messages originate does not stop the warning being issued. A simple way is to include Task.Delay(0), which causes no delay but supresses the warning, but this would be considered a serious “hack” so it has not been done.

### Problems with ‘Async’ Code

Problems were experienced with the model load. It was being populated with the sample data before the deserialised data could be added. This was because the Load method was being called from the Music Collection constructor. The load methods needs to be ***async*** because it contains calls to async methods accessing the XML file. This meant that the call could not be awaited in the constructor as it cannot be marked as async, resulting in the check for not data returned passing and the sample data being loaded before the source file is deserialised. Various searches on the internet also came across various warnings about calling async methods from constructors being potential sources for deadlock and this is not good practice.

The solution was to move the LoadAsync to be consumed by the PersistenceRepository, which is launched from within the App.Xaml.cs. The persisted file is read and the model loaded from within the "OnLaunched" method of the App.Xaml.cs and the current model is persisted to the underlying XML file from within the "OnSuspending" method again within the App.Xaml.cs file. Ie at application startup and application shutdown.

This is a nicer separation of concerns, and also means the load methods can be entirely Async, which allows them to be properly "awaited" and therefore the app runs correctly. The model deserialises correctly without interference from the loading of sample data, as the deserialise process completes before this check to load the sample data is reached.

## Binding to and displaying images from the User (Known) folders.

Various problems were encountered when binding the UI elements to image files. The must be bound to with Uri’s that are loaded at runtime with BitMapImages. This is fine, with images included within the project. However, when the image is located within the users’ folders, even though the application has the capability to access the users folders, the image file must be accessed by reading it with an IRandomAccessStream and creating the BitMapImage from that.

This is OK, but a better solution which also avoids the need to save image files, is to get reference to files from the LastFM album information and use those always. This avoids the need to store images locally and the need to read them. Asynchronous loading is easier with urls too.

This involved calling the LastFM service to get image url’s for audio files when they are loaded. This is a more consistent approach

## Design Time Data in Windows 8 Store (Xaml)

Design time instances should be kept entirely within the UI application layer. Moving them to external assemblies or projects appears to cause errors stopping the data appearing in the design time xaml pages.

In the case of MVVMLight, it is completely possible and acceptable to have design time resource instances injected using the SimpleIoC, again just make sure all references down the design time path are local to the UI project.

However, the current project works fine with the images for the ItemPage.xaml, and the SplitPage.xaml. However, if the design time data, which is coded within the Design folder of the UI project, is activated for the ImportLastFM.xaml it stops the ItemPage and SplitPage from working.

This needs further investigation to see why, but there is not enough time just now. Considered a minor problem and the ImportLastFM.xaml uses the same User control as the SplitPage (except it doesn’t include the Media Player control) so the control is tested.

## Problems with Playlist only including certain tracks

When supplying the track and associated albums belonging to the selected Playlist on the Group details page, it was having the effect of deleting the tracks that were not in the playlist, permanently from the Album.

This was because the collections returned from the queries are only references to the corresponding albums in the domain model context. I.e. they are shallow copies.

The solution was to have a completely new reference to these albums, which could safely be updated with only the tracks belonging to the Playlist. This was achieved by creating a Clone method in the Album class and providing a shallow copy, references to all child classes, except the tracks. They are initialised. This gives a completely new reference to the album though, so does not affect the original album instance in the domain model context. The other child classes were safely left because there is no attempt to update them.

This could be better designed by separating the Playlist from the Album, but then it wouldn’t fit nicely into the Summary page, which is by far the nicer way of dealing with playlists.

## Use of TagLib# and Windows 8 Store app

Taglib# makes use of calls that have been restricted in the Windows 8 Store environment, ie they are not provided. There is a workaround documented within the code of the MusicFileHelper class to code in such a way as to avoid making calls to the banned api’s. The source of the workaround can be found <http://stackoverflow.com/questions/13381494/c-sharp-windows-store-app-granting-capabilities-to-used-dll-s>.

## Unit Testing

There are no unit tests within this project. This is not deliberate, just that it hasn’t happened that way. I spent a significant part of the project trying to get the xaml based UI layer to work without enough time to implement unit tests for the ViewModel.

# Evaluation

This solution contains a lot of development, even though it doesn’t appear to be a big project.

This is the first time I’ve used most of the technologies included, which therefore required a steep learning curve.

It is also the first time I’ve really had to code an asynchronous application and I don’t think I gave quite enough consideration to this early enough in the development. I did immediately decide to opt for the Task based model, along with the potential use of the ConcurrentBag to deal with the concurrency issues in multi-threaded application. The task based model is the correct choice, but upon further investigation later in the project the use of the CuncurrentBag was considered inappropriate. I don’t think I’ve coded the best approach to the asynchronous and concurrent programming model, but it’s a reasonable start, one that has highlighted some of the issues that must be considered.

I also think my choice of domain model, with Entity Framework style navigation properties was not the best decision ever made. In my defence, Entity Framework has featured heavily in most projects throughout this degree course and my mind was set in that pattern, hence the model. However, it still works well and quickly, despite the more complex code for deserialising and updating the model.

It has suggested the need to become acquainted with the likes of SqlLite for persistence. The other thing to consider, is probably more in keeping with the intention of these Windows 8 Store apps and that is the use of Cloud based storage for information. This means the app becomes a consumer of web services, not necessarily a bad idea.

As far as the technology is concerned, there are still many facilities lacking, compared to desktop development. It is also still the situation that compatible 3rd party tools is quite limited again restricting development.

The use of MVVMLight was a significant improvement over the last coursework in providing the Ioc as well as the MVVM pattern. MVVMLight, provides so much more functionality, things like messaging which can be used to communicate between pages as highlighted previously. It would be worth spending time to become acquainted with these features, as it seems to be a very complete framework and becoming quite popular.

# Further Development

## Code Refactoring

### Redesign updating of UI when adding audio files.

The design should be changed to update the UI automatically as each audio file is added, therefore keeping the UI more responsive. There is a partially coded solution to this in the code, of the MainViewModel in the ImportMP3Handler. It has been left commented out as it is not complete.

### Caching the LastFM calls

Again in the process for adding audio files, the LastFMService should be coded to cache the calls to the web service. This would avoid the need to repeated make calls for the same information when adding multiple audio files belonging to the same album. Obviously only successful responses should be cached. The cache wouldn’t need to be that big but would carry a significant improvement in performance.

### User Feedback when calling LastFM and the MP3 import facility

The “process working” feedback, the circle of dots, must be included on the Import pages to signify that something is happening.

## Known Issues and Bugs

### Trap error if XML file corrupt

It is possible that the XML file could become corrupt which causes the deserialization to throw an exception. It is currently unhandled, which should be altered as soon as possible. The solution is to trap the error, and delete and recreate the XML file. It affects the PersistencyRepository.LoadAsync method. It is not likely to happen as there is no functionality to delete albums so the file is only likely to grow in size. Development issues caused the file to “shorten” which showed up the error.

### Design Time Data

Design time data has been setup for each of the pages in the application. There is currently a bug that when the ViewModelLocator is configured to inject the DesignLastFMModelDataService into the LastFMViewModel, it stops the design time data being displayed in the ItemPage and SplitPage within the Xaml editor. It doesn’t matter in what order the services are injected either.

This has had to be left unresolved due to running out of time to complete the project. It would obviously be investigated fully as it should be possible to resolve it.

## Future Development

### Add Playlist functionality

The current version of the application allows for album tracks to be displayed within Playlists. However, due to time constraints it was not possible to provide the facility to create playlists. This would be completed as future developments. The requirements of the coursework are not compromised without this functionality which is why it’s not included

### Search Facility

There is currently no search facility of the Music Collection. It doesn’t detract front the current application as the collection is browsed from the main page either by Artist, Genre or Playlist and within each the entire collection is visible.

This would be best served in the Windows 8 store environment by implementing the search contract, which then makes the application searchable from outside it. Again the requirements of the coursework are not compromised without this functionality which is why it’s not included.

# References

### Sources used researching the use of Concurrent collections:

Confirms that ConcurrentBag can be a source of Memory Leaks

<http://stackoverflow.com/questions/8293297/how-can-i-free-up-memory-used-by-a-parallel-task>

How to use the dictionary

<http://stackoverflow.com/questions/8225775/concurrent-dictionary-correct-usage>

Best Practices for ConcurrentDictionary

<http://arbel.net/2013/02/03/best-practices-for-using-concurrentdictionary/>

MSDN's documetation for Concurrentdictionary

<http://msdn.microsoft.com/en-us/library/dd287191.aspx>

### The workaround for Taglib# usage in Windws 8 Store app

<http://stackoverflow.com/questions/13381494/c-sharp-windows-store-app-granting-capabilities-to-used-dll-s>

### Reading images from user folders

<http://social.msdn.microsoft.com/Forums/en-US/winappswithcsharp/thread/9c9dc5b3-0b64-404a-ac15-ddddc22004a6/>

<http://www.devcurry.com/2013/02/windows-8-apps-creating-picture-library.html>

<http://mtaulty.com/CommunityServer/blogs/mike_taultys_blog/archive/2013/01/25/windows-8-making-a-simple-photo-viewer-in-c-and-xaml.aspx>

### Build a decoupled message based way of sending parameters.

<http://geoffwebbercross.blogspot.co.uk/2012/04/mvvm-light-passing-params-to-target.html>

Using Properties and parameter strings: Mr Laurant Bugnion (Galasoft: creator of MVVMLight

<http://blog.galasoft.ch/archive/2011/01/06/navigation-in-a-wp7-application-with-mvvm-light.aspx>

MSDN article; Mr L Bugnion again

<http://msdn.microsoft.com/en-us/magazine/jj651572.aspx>

A good CodePlex example, shows the command and parameters.

<http://www.codeproject.com/Articles/456558/Getting-Started-w-Windows-8-MVVM-Light-and-EventTo>

A couple of StackOverflow questions

<http://stackoverflow.com/questions/10971023/page-navigation-through-viewmodel-using-mvvmlight-in-windows-8>

<http://stackoverflow.com/questions/11710859/implementing-state-handling-in-windows-8-using-mvvm-light>

1. This is dependent on my translation skills. I wouldn’t give tuppence for the translations but the intention is there. [↑](#footnote-ref-1)