Task 7C – Spike: Data Everywhere

Creating Data Driven Mobile Applications Jai Lafferty 101610772

Goals:

> This section is an overview highlighting what the task is aiming to teach or upskill.

This task aims to demonstrate the ability for an app to read data from an API before displaying it to the user while offering the option of downloading the data into persistent storage for offline use.

> The following list outlines the goal broken down into more specific knowledge gaps involved in the goal.

- Setting Up Core Data
- Understanding Completion Handlers

Tools and Resources Used:

> This section lists related software, tools, libraries, API's, and other resources used for this knowledge gap.

- XCode
- Canvas Resources
- Previous Tasks
- iOS Developer Documentation

Knowledge Gaps and Solutions

> This section presents the listed knowledge gaps and their solutions with supporting images, screenshots and captions where appropriate/required.

Gap 1: Setting Up Core Data

Core data was far more easier than I imagined to both set up and use. I used an amazing article on Medium written by Shashikant Jagtap entitled Core Data with Swift 4 for Beginners

It took me through the bare basics of adding, saving and retrieving data from the persistent database. I planned on using Core Data to store very basic information about the albums downloaded through the iTunes API.

 Entity Relationship Diagram of my Core Data stack

Note the one-to-many relationship between Album and Tracks, this allowed me to have a collection of tracks pointing towards the same album, keeping the integrity of the previously memory-based storage solution with structs and arrays.

Actually implementing this, I created a class called CoreDataController whose role was obviously to control all Core Data related methods and features.

 Core Data Controller Model

Note: memAlbum and MemTrack denote the Album and Track structs stored in memory

Actually inserting and fetching data was relatively simple with a few lines of straight forward code.

```
func writeAlbum(album: memAlbum) {
    let newAlbum = Album(context: context)

    newAlbum.title = album.title
    newAlbum.itunesId = album.itunesID
    newAlbum.imageUrl = album.imgUrl

    do {
        try context.save()
    }catch {
        print("Failed to save album to core data")
    }
}
```

Writing a memAlbum to Core Data

I found that you can treat Core Data Objects just like the structs I was using before with the only difference being having to Save once loading the data into the object was complete.

Fetching was slightly different in the fact that you have to structure your query before running it, there didn't appear to be a one line 'return all Album objects' statement.

```
func findAlbum(collId: String) -> Album? {
    let request = NSFetchRequest<Album>(entityName: "Album")
    // Filtering by collectionID
    request.predicate = NSPredicate(format: "itunesId == %@", collId)
    do {
        let result = try context.fetch(request)
        let album = result[0]

        return album
    } catch {
        print("Failed to find Album from Core Data")
    }
    return nil
}
```

Returning an album with a given collection ID

I then spent a significant amount of time researching how to go straight from core data to a TableView. Realising I was spending too much time looking at NSFetchedResultsController and trying to wrap my head around it I figured I'd go down the more expensive path and load all relevant data back into memory before displaying it in my table view as you conventionally would. This is why there's the need for the conversion methods in my CoreDataController object.

Gap 2: Wrestling with Async Tasks

I struggled in the last iteration of this app with getting views to appear once leading was complete. Shamefully I placed hard coded time delays into some segues to allow the app to load the data. Knowing this was wrong and aiming to rectify the mess I created, I decided to read up on how to tackle these tasks.

In my implementation, the API calls and the parsing of the json was done in a single asyncronous call. This allowed me to add ActivityIndicators to the UI displaying to the user that something is loading.

 Loading indicator

As the next view requires the parsed data from the API call, the main thread waits for the task to be completed before performing the segue.

Due to the small amount of data actually being downloaded this isn't a very long wait for the user, typically around half a second. However, should the user wish to download a large set of albums, this could take a lot longer.

Open Issues and Recommendations

> This section outlines any open issues, risks, bugs, etc..., and highlights potential approaches for trying to address them in the future.

Duplicate Albums

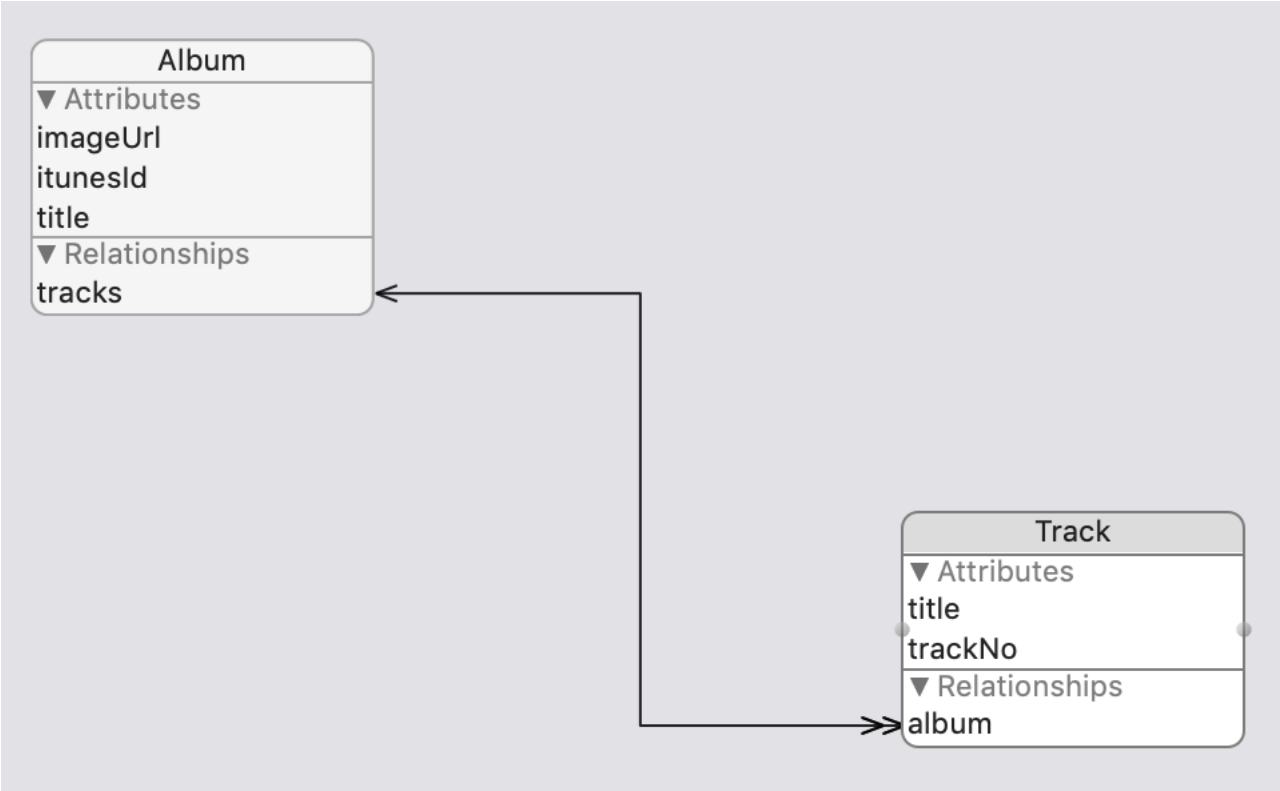
This problem still persists from the last iteration of this application, different versions of the same album under the same name give the impression of duplicate albums in the tableView.

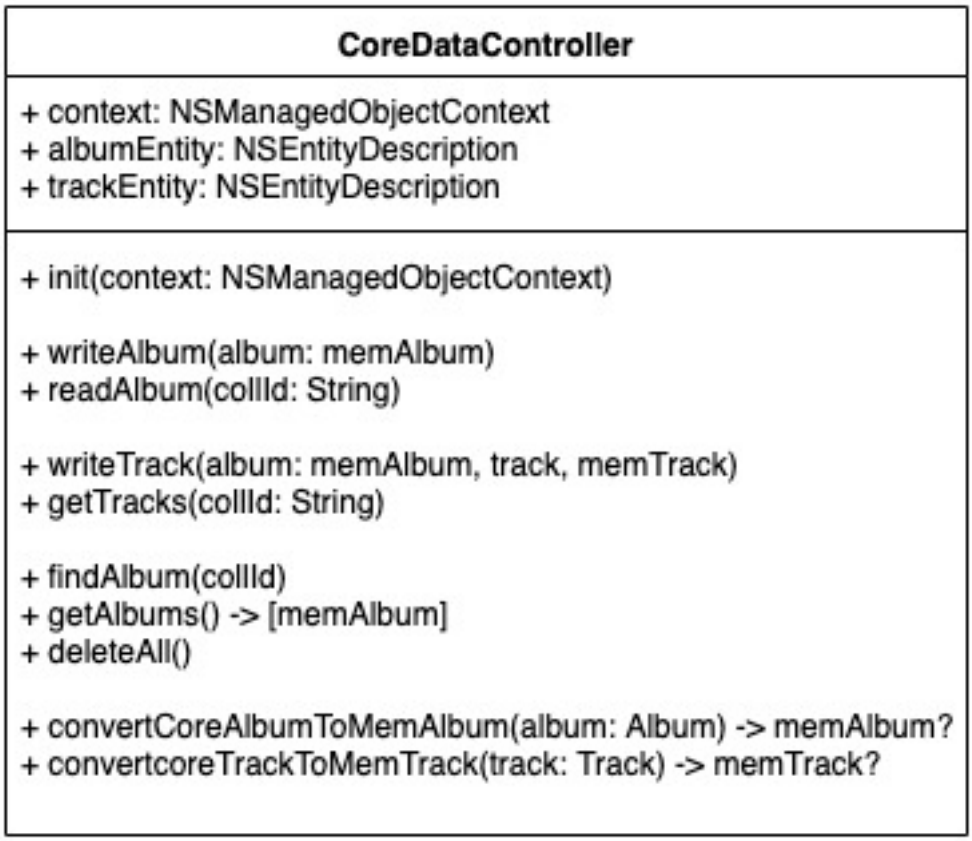
Singles Loading Bug

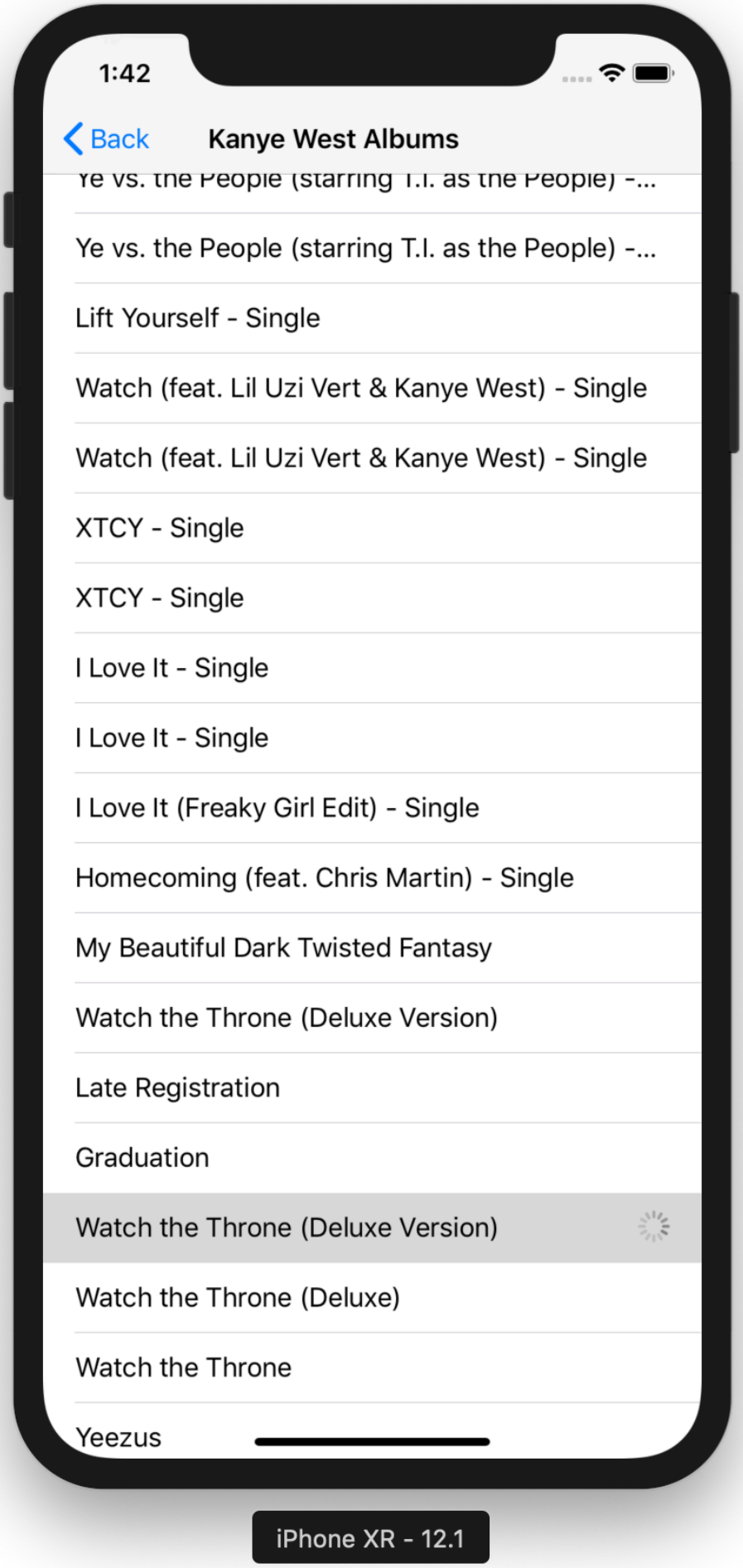
Some singles (albums with typically one track) load too fast for the aync task, resulting in the details screen being empty of tracks. It appears that the segue occurs almost at the same time as parsing completing perhaps parsing isn't 100% complete at the time of the segue being performed.

Possible Modifications

- Some sort of filtering function that allows each album to only show the one with the most tracks - defaulting to 'Deluxe' albums
- Allowing the user to select an artist, only thing needing altering is the ability to search for an artistID through the iTunes API.


Album
▼ Attributes
imageUrl
itunesId
title
▼ Relationships
tracks
Track
▼ Attributes
title
trackNo
▼ Relationships
album


CoreDataController
+ context: NSManagedObjectContext
+ albumEntity: NSEntityDescription
+ trackEntity: NSEntityDescription
+ init(context: NSManagedObjectContext)
+ writeAlbum(album: memAlbum)
+ readAlbum(collId: String)
+ writeTrack(album: memAlbum, track, memTrack)
+ getTracks(collId: String)
+ findAlbum(collId)
+ getAlbums() -> [memAlbum]
+ deleteAll()
+ convertCoreAlbumToMemAlbum(album: Album) -> memAlbum?
+ convertcoreTrackToMemTrack(track: Track) -> memTrack?



iPhone XR - 12.1