WhatToWatch

Introduction

Most often than not, if not all the time, one is encountered with the dilemma of not knowing or having the most optimal answer for probably the most important question that surfaces when wanting to watch something: what should one watch. Whether or not one is accompanied or just alone trying to kill some time, this question always resurfaces and always will. Too much time is spent browsing, searching, discussing (and these verbs are always sprinkled with a huge amount of arguing to the brink of actually fighting. Before the company that desires to watch a movie or show even knows it, too much time has passed hence it being too late for beginning to watch something. All that is left is tears and broken (relation-)friendships. Therefore, the app *WhatToWatch* exists. To eliminate the mentioned problems and bring ease to the decisions of choosing what to watch.

WhatToWatch presents completely randomly chosen movies, and solely movies (shows excluded) to the user of the app. The user is then able to either like or dislike the presented option. Upon liking the alternative, it is added to a list where all the other liked movies can be found. The same goes for disliking alternatives. The disliked alternatives will not appear again in the random pool if the user does not wish to since the list of all the disliked movies can be reset. An individual movie can also be reset, perhaps an event of an unintentional misclick took place. Every alternative, after either a like or dislike, can be added to another separate list that goes under the name *Already watched*. The name insinuates the exact functionality of this list - the user of WhatToWatch can, to every alternative presented, choose to add the movie to this list indicating that it has already been watched. The movie will not be presented again in the pool if the user chooses so since there will be an alternative for the user to reset this list also. Information such as a description over the story of the movie, a cast list, genres will also be available for every movie. In this way the user can get a slight grip of what they are dealing with.

Figure 1: The start page when starting the application.

Figure 2: The view where the user can either swipe on the movie poster or tap the like or dislike button depending on if the user finds the movie interesting. Swiping right indicates a like while a right swipe indicates a dislike. Upon a tap on the information icon, information about the movie will be displayed.

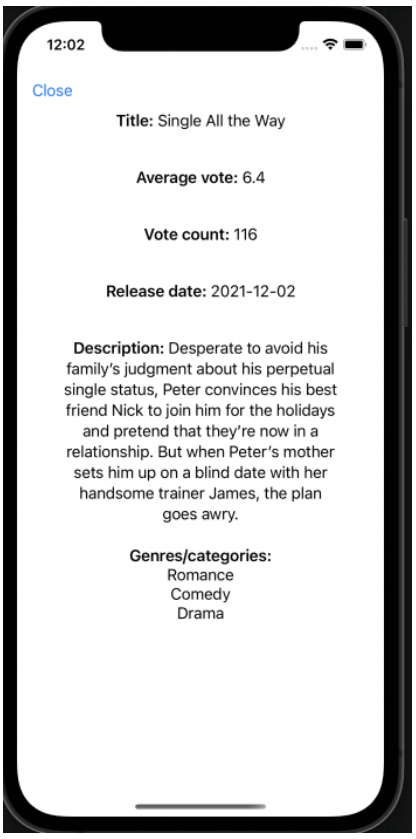

Figure 3: Information that is displayed upon a tap on the information icon.

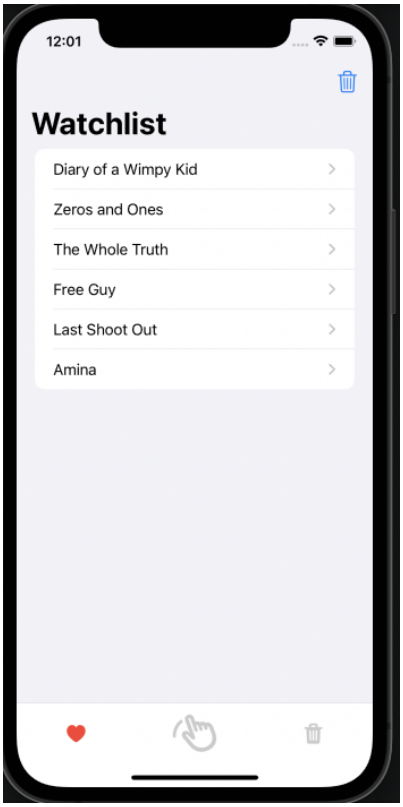

Figure 4: An instance of the contents of a Watchlist (movies that the user liked). There is also one for the Trashlist (movies that the user disliked) that is identical to the Watchlist.

Architecture and planning

First the approach of where the data of WhatToWatch has to be addressed. WhatToWatch uses an *API* (application programming interface) provided by *The Movie Database* (TMDB, https://www.themoviedb.org/). However there are several steps that have to be done before being able to use the API. An account has to be created at their website. Furthermore an inquiry has to be done and the organization TMDB will then respond with a corresponding (to the account created) unique API key which is then used as an identification in order to be able to use the API and all its data. When these steps are done, the overall design of the app has to be implemented. This is done to give a general idea of where the data will be injected and visible. Furthermore the different files containing the models and decoders for the *JSON-files* have to be implemented in order to get a grasp over how the data is going to be injected into the app. The general idea of the usage of data is to not use Core Data but to diligently utilize different API calls in the data models of the application hence the TMDB provides an unlimited amount of API calls to their website. Finally, different tests of the program are going to be implemented. And if excess time is given at the end and the app is fully functioning, finishing touches are ought to be done e.g animations.

Implementation

Design and data

The implementation of WhatToWatch pretty much went according to the initial desired *Architecture and planning* (as described above). There were no major deviations from the planning that are worth mentioning. An account was created at TMDB. Furthermore an API key was requested and ultimately provided giving full access to the data.

Later the design of the overall application was, with reservations for changes, implemented. The implementation in terms of the design of WhatToWatch had to look as close as the finished product right from the start. This was done since it was perceived to facilitate the further development of the application and injection of data. However, reservations were made for potential changes.

Since the project was due to be implemented by a duo, one had the assignment of designing the app as a whole in order to direct where the data ought to be injected. The other one implemented some of the initial data models, but not all of them, in order to test out the API calls and data. When these steps were done the duo had the task of implementing the remaining data models and data injections.

Core Data

Core Data was not used in the implementation of WhatToWatch since it was not considered necessary. All the different dynamics of the application were implemented solely with API calls to TMDB. It was also considered by the duo to be further knowledge enriching in using API:s and API calls. However, later on in the development difficulties with this strategy were encountered. For instance, one challenge that really made one stuck was how to render a list of a user's *watch list* (list over the movies the user finds interesting and desires to watch) movies by only knowing the specific *movie ID*. The ID:s of the movies were saved in the property wrapper *@AppStorage* using an array of integers. A whole movie as an object could have been stored in @AppStorage although it is not optimal. @AppStorage should not contain such *heavy* contents but must often only references/attributes of objects. Multiple API calls had to be made with the movie ID:s as a parameter. In the beginning only the latest movie added to the watch list was fetched accordingly. This led to the developers making a change to one of the data models and the *remote* that handled the JSON-data. Instead of only using one remote where only one API call could be handled, a modification to the one of the data models and remote had to be made making them taking in an array of remotes making it possible to handle multiple API calls instead of only one. With Core Data every movie object that a user wishes to have in a watch list would be inserted as whole in a Entity to moreover be fetched and displayed (in contrast to making an API call with only the movie ID at use).

```
@AppStorage("likedMoviesList") var likedMoviesList = [Int]()
@AppStorage("dislikedMoviesList") var dislikedMoviesList = [Int]()
@AppStorage("counter") var counter = Int()
@AppStorage("genres") var genres = [String]()
@AppStorage("score") var score: String = "1"
@AppStorage("hasSavedFilter") var hasSavedFilter = Bool()
```

Figure 5: A showcase of how data is stored using @AppStorage. For instance, the likedMoviesList and dislikedMoviesList contain and hold the ID:s for the specific movies so that an API call with these specific numbers can later be made.

One could argue that Core Data is much easier, nevertheless itself has a learning curve that takes time to get accustomed to. The duo did not consider the one or the other to be easier or harder nonetheless demanding different ways of thinking and approaching problems to furthermore find the solutions to them. Despite that it was reflected on and led to the conclusion that using API calls were more optimal right now and also could be done.

Filter system

Initially there was no intention of implementing a filter system for the movies. However, with the addition of time in the end a decision of creating such a system was made. Furthermore, this led to the implementation phase being a bit more complicated in contrast to previous additions of features. The filter was put in place and is fully functional nevertheless, the code was perceived to be crammed in and not fully refactored in. Since the filter system was considered late and its implementation was a bit harder than the other features it demanded more time and consideration and also careful consideration on where the code should be placed. More on this matter regarding the implementation of the filter system will be elaborated on later down below.

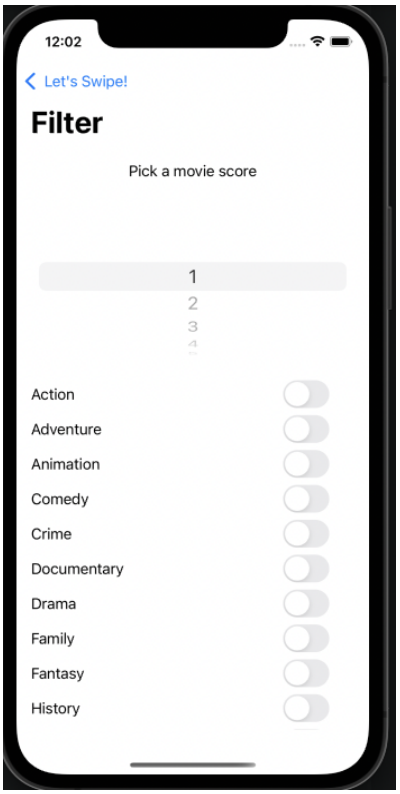

Figure 6: A clipboard of the Filter view.

Delete function

The ability for the user to delete movies added to either the Watchlist or Trashlist was a big challenge for the project. The commencing version of the execution of said delete function deleted the designated object, however, it did not render it properly. The non-targeted object at the absolute bottom of the list (Watchlist or Trashlist) appeared to be removed and was no longer visible. Even so, upon refreshing by leaving the tab and/or view the data was properly loaded back in and fetched. This led to the lists properly showing all the remaining movies

excluding the one that the user initially wanted to delete. Eventually, this was also fixed meaning an exit out of the tab/view was no longer necessary since the list was rendered properly. More on this and the challenges and corresponding lessons learned it brought with it will also be elaborated more later on.

Learnings

API:s and API calls

A lot of new things were learned during the development of the project. One instance of the stated sentence is the further use of API:s, API calls and remotes. There has been a bit more advanced development of understanding of how they all interconnected, how they work together and how one can utilize them. Another instance of new learning is the understanding of how one can work with storing data without having to use a database. There are other alternatives to Core Data since Core Data is a sort of a database. Most often when one talks about data and storing it one automatically thinks about databases. The same goes for fetching and deleting data. Furthermore, an application does not always have to use a traditional database to store specific data. An alternative to databases could be working with API calls as has been done with WhatToWatch. There will always be the need to store data in some shape or form however, it does not always have to be done in the traditional way.

Data models and fetching data

Another major new learning was the instantiating of data from different data models. Previously data was passed along multiple views as parameters in Navigationlink. This complicated and confused the development process in an unnecessary way that could be avoided easily. In addition to this matter, a minor restructure of how the data was fetched was made. Data was instantiated, fetched and used in only the more suitable places in the program in contrast to, as it was done in the early development phase, doing it in the very beginning and in the places in the program should have been done

There has also been, as previously mentioned before, a deepened understanding in how one can make multiple API calls at once in order to perhaps render an desired list of items. Other new knowledge acquired during the process of developing WhatToWatch are getting a glimpse at how a fraction of industry development *might* look like when developing a product. This concludes dividing the work into different parts and actions (planning, drawing the initial design, assigning one partner to do the design while the other one is responsible for the first data model), discussing with the development partner, talking to course mates,

asking course mates for the right type of help, providing course mates with the right type of help, trying to follow design patterns that were learned in previous courses, working in a team although the team only consist of two persons and ultimately working with git and GitHub while being on a team instead.

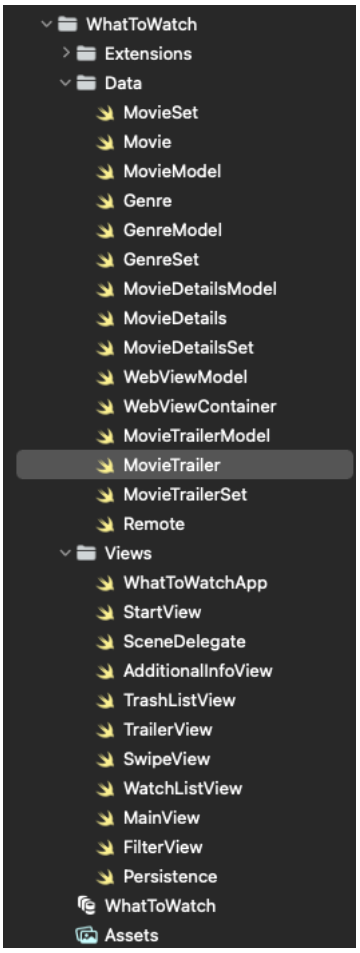

Figure 7: A demonstration of the different files, how they are structured and their responsibilities.

Filter

As mentioned above, the implementation of the filer system for WhatToWatch also brought with it its own wisdoms. By attempting to create said functionality it demonstrated how different states of the program had to be passed along and further challenged the already established understanding of usage of API:s and API calls. Also, sometimes the filter did not render movies following the set filter and sometimes the filter rendered the same movie twice. After a lot of attempts it was finished but the vision of constantly being able to improve on it will always be present.

Delete function

The first version of the delete function deleted the selected object but rendered it improperly by removing the last item at the bottom of the list (Watchlist and Trashlist). However, when one refreshed the view, forcing the application to make an API call to fetch the data again, it properly showed the list with the originally intended object gone. This was solely a visual but major bug. One should not have been forced to leave the view in order to let it refresh for the sake of displaying what was removed and not. This edition of the delete function turned out to be missing a removal in the remote also, not just the list. By only removing from the index in the list the specific movie ID:s for each movie shifted in an unusual manner leading to a visual bug. By also removing the specific ID from the remote also remedied this matter.

Restructuring and refactoring of code

One of the, if not the, biggest learnings the implementation of WhatToWatch brought with it was in how many different ways code can be written and refactored in. It is not always about writing about functional code but also beautiful code. This is something that is overlooked very often. This also demonstrated how, by refactoring code and making it, not only more optimized, but more visually aesthetic, can facilitate the development in general. This is a lesson the duo will carry with it going into future projects.

Outlook

WhatToWatch is an ambitious project and application and it is desired to build upon it even after submission. This product means a lot to the developers and it is aspired to further develop it. Some visions for the future is to add some more functionalities to the application. One instance of this is the opportunity for users to create a new section where the users can add the movie that has been added to the watch list to a *Already watched* list in order to give the user the freedom to keep track of which movies have already been seen. The same goes for the movies that have been added to the trash list, they too should be able to be added to this newly created list. Eventually and ideally, another goal is to publish the product to the *App Store* since this is a product that has a lot of faith in it and is considered useful and needed.