Student Number - 140150025

Notebook report

Assignment 2 – SEM2220 Mobile Solutions

# Introduction

This report is a write up of the second assignment for SEM2220 Mobile Solutions. The assignment was to create a native mobile application in either Java for Android or Swift for IOS. In short the application needed to be able to take notes, search Guardian content using the Guardian API [] and link articles to the notes previously taken.

The application was written in swift using XCode. Some swift code was provided to help access the Guardian API. This code was adapted to aid in finding a solution to the assignment.

This report will cover the following:

1. How the assignment was implemented in Swift including problems encountered.
2. Testing procedure and results
3. Evaluation of the solution

# Implementation

I began by looking at the previous workshops that SEM2220 had provided to refresh my memory of how to use XCode and program in Swift. After scanning through the workshops and workshop solutions on GitLab [], I found the solutions that would aid me in completing the assignments based off of the function requirements shown in Appendix 1.

## Note Functionality

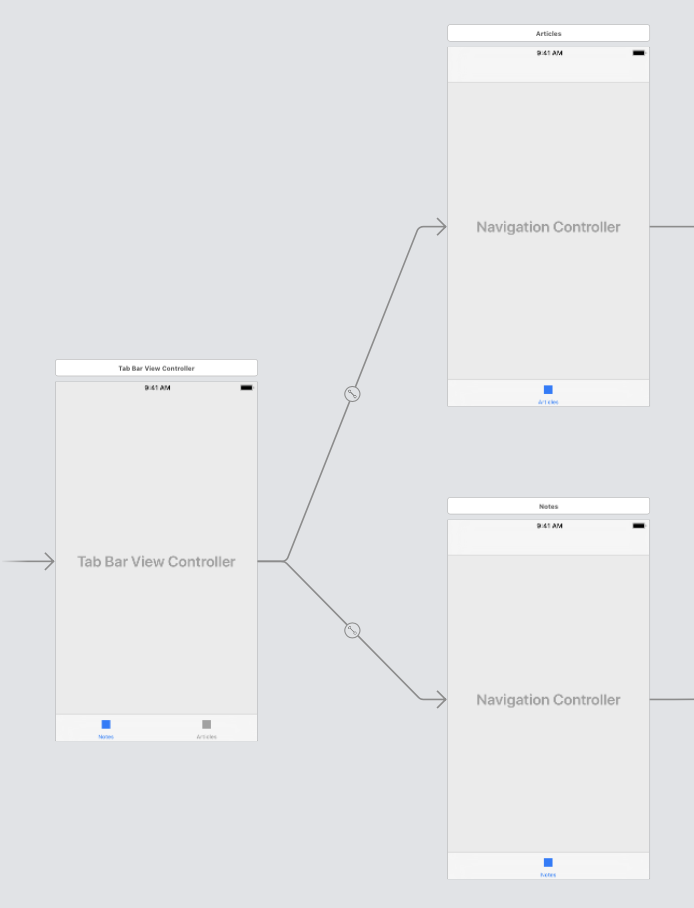


Figure 1 - Tab Bar Controller connecting to Two Navigation Controllers

I started by creating a tab bar view controller, a navigation controller being the initial view, with one of the tab views to be for creating and viewing notes. I used a table view controller to view the notes that were going to be created. I noticed I wasn’t able to add any bar button items on the navigation bar, I moved the navigation controller and tab bar controller around and was able to add the item then, this is why two navigation controllers have been used in the final solution as shown in Figure 1. A segue was created using the bar item to a new view controller that was to be used for creating new notes and storing them.

I decided to use Core Data as the storage solution it was covered in the IOS workshops and very easy to implement as it is built into XCode.

The implementation for creating a new note is straight forward as it is using techniques to store the data as learnt in the IOS workshops, the only difference is that later I would implement an edit note function that would be triggered if a *Note* variable was passed to the controller, if a *Note* wasn’t passed to the controller then a new note would be created.

When the notes were shown in the table view, I decided to create another table view controller that would be shown when a user clicks on the note itself. Clicking on the note would pass the details using a *prepare* function to the new controller. When creating the UI for the controller, I used a vertical stack and text views for the title and the table would be used to show the URL’s linked to the note.

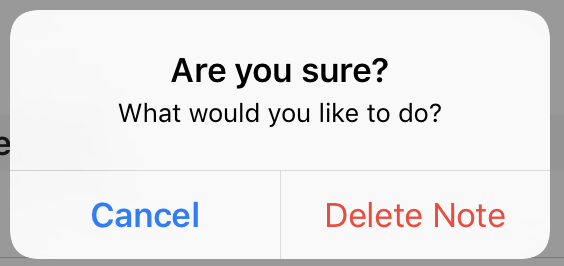


Figure 2 – UIAlertController to ensure user is selecting the correct option when deleting a note

Later into the assignment I added the ability to click and delete the URL’s listed. Clicking the URL would show the user the details of the Article linked to the note. Two buttons were added, one being *edit* which was mentioned earlier, this would use the create note UI and controller by passing a note to it. The other button being a delete note function. As the button was relatively close to the edit button, I added an *UIAlertController* following a tutorial [http://swiftdeveloperblog.com/code-examples/create-uialertcontroller-with-ok-and-cancel-buttons-in-swift/] online to make sure the user was selecting the right option.

Implementing the note functionality was fairly straight forward. The GitLab solutions helped massively and helped me have a better understanding of how Swift and XCode work.

## Search Functionality

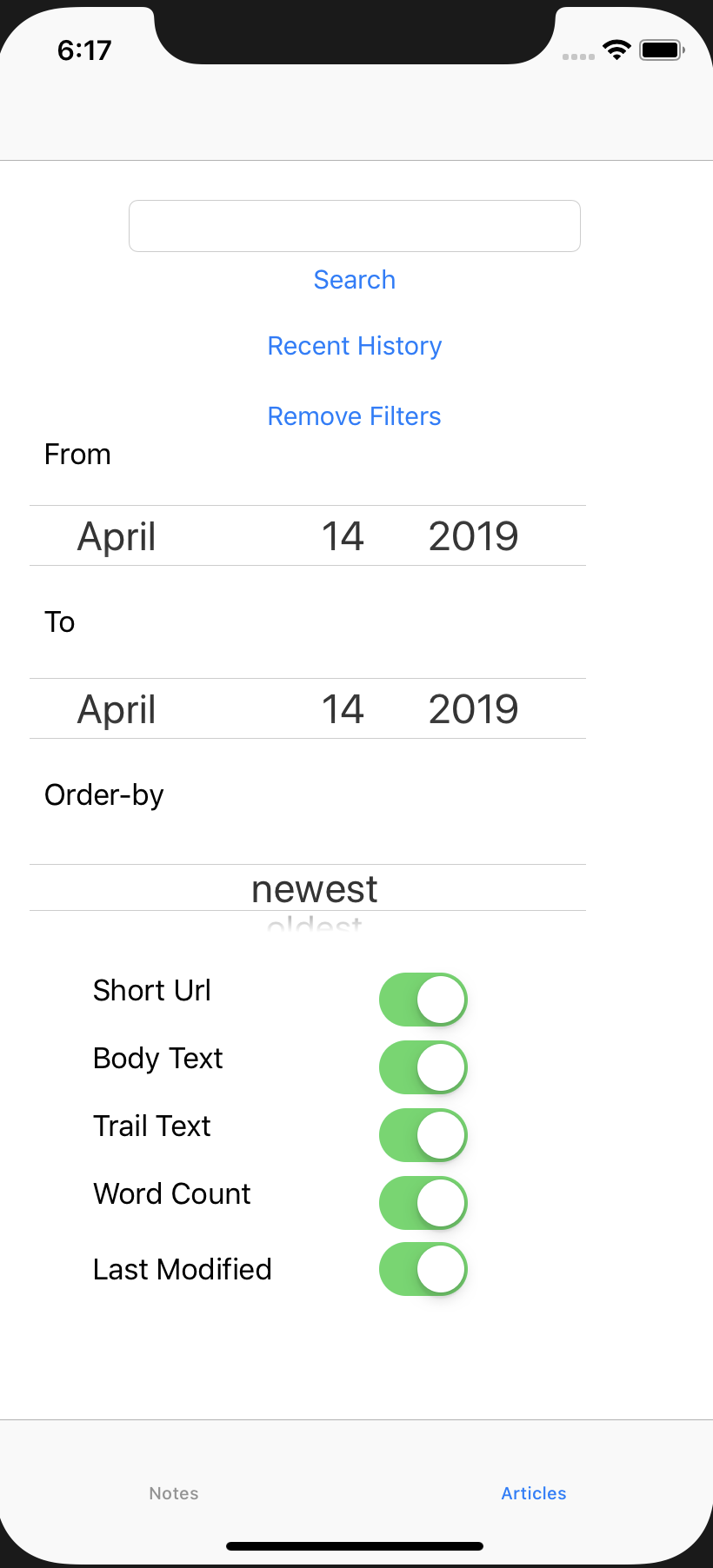


Figure 3 - Search UI

I used the code provided on Gitlab [] to communicate with the Guardian API. An example was shown on how to get a response from the API with a search term. I based the start of the search controller on this example. I created the UI, shown in Figure 3, with the additional filter controls as described in Appendix 1 FR-3a.

The UI used a spinning mechanism called a *Picker* which allows the user to pick an option from present values. To use the picker the values had to set programmatically, I used a tutorial online [https://codewithchris.com/uipickerview-example/] to help implement the necessary functions. The date picker was easy to extract data from as this didn’t require additional functions and setup.

Linking the filters to the search was easy as filter data types (*GuardianContentFilters*) just needed setting and passing to the provided Guardian client. The show-fields filter (triggered by using *UISwitches* shown in Figure 3) created a comma-separated list that would be passed to the client. The show-fields filter caused issues as when they were requested, an error would occur stating that the response was in the wrong format. This issue would soon be resolved when Neil Taylor released a bug fix for the provided code.

I created a class that would check for an internet connection and ping the Guardian API when the search button is pressed. A *Completion Handler* was used to decide what to do with the result. If no connection was present, it would alert the user to use the cached content, otherwise the API would be called, and the results would be cached and shown to the user. The search would also be stored in Core Data to be used for recent searches.

The Core Data *RecentSearch* entity takes the search term, the filters used, and the key used to store the results in the cache.

Caching the content saved from results was a difficult task for me. I began to look at methods of caching that didn’t use Core Data. I found *NSCache*, which I spent a considerable amount of time implementing, this worked but would throw errors when trying to retrieve data at certain times. It was later discovered that *NSCache* is only temporary, and the cache is removed once the application is closed. I then started to look at *NSKeyedArchiver* which can encode objects and store them into a file. I built a class that would take the *GuardianOpenPlatformResult* data type and the class used *NSCoder* to encode/decode the data. After a large amount of time was spent trying to get this to work, I resulted in editing the *Completion Handlers* in the code provided to pass the raw JSON and store this in the *UserDefaults.*

The recent searches would follow use the same functionality as the search function did. First checking if the phone has an internet connection and pinging the host. If the connection is up, the search will run again. If the connection is down, the user can click on recent searches that have the content cached. This was easy to implement.

The results of either a new search or of a cached search were then displayed on another table view controller. This was easy to implement as it had been done a number of times before. It allows users to click on the article and view the details about the article including the details that were requested from the *show-fields* filter. The detail page was implemented using an array containing the order in which the details about the article should be presented. This array is manipulated upon loading of the screen and the fields are displayed.

A button was added to the article detail page that would allow the user to assign an article to a selection of notes. I adapted the example code provided on GitLab that allowed for a multiple selection list on a table view, but the majority of the code remained the same.

# Appendix

Appendix 1

|  |  |
| --- | --- |
| Requirement No. | Requirement |
| FR-1 | A user can enter notes about a topic. A note has a title, a date it was created, a latest modification date and the text for the note. |
| FR-2 | A user can search for links to content for specified search terms the Guardian’s Open Platform API. From the returned results, it is possible to select a number of links and associate them with notes that are created in FR-1. The app should store the URL and the title for the links. |
| FR-3 | A user can enter search terms and apply filters to configure what is searched for and what is in the results that are returned.   1. As a minimum, the user can enter a free-text term as a search term and the user can specify the date-from, date-to, order-by and show-fields filters that are passed in the query to the API. 2. Optionally, for a more advanced project, the user can use Boolean operators, ( ) brackets and “ “ quotations to enter more advanced queries. This could be specified as a string, but a better implementation could provide a way to build that query using the UI in the app; this could help reduce the errors in more complex queries. Further, a more advanced project could support filters for production-office, section and tag. |
| FR-4 | A user can look through lists of recent searches for content in the API and run the search again. |
| FR-5 | To support offline access to information, the app can cache detailed article information, e.g. headline, body text, standFirst text, trailText and wordCount. These details should only be kept for up to 24 hours. The app needs to delete content older than 24 hours. |
| FR-6 | A user can view a note and its linked content from the API. By selecting one of the linked content items, the content is displayed in a web browser. |
| FR-7 | A user can search for notes that match specific text. |