iOS Project Report

Laureen Schausberger

PROJECT-REPORT

submitted for

MSc Mobile Application Development

in Canterbury

December 2018

This report was created as part of the course

iPhone Application Design

during

Fall Semester 2018

Advisor:

Dr. Christos Efstratiou

Contents

Chapter 1

Introduction

The following chapters will give an overview of the iOS application, which was created during the iPhone Application Design, will mention integrated features and explain the main implementation details.

The goal of this project was to build a complete iOS application, which views dynamic data from an API. The developed app should be able to demonstrate the skills learnt in this module.

Even though it was suggested to use the Flickr.com as an endpoint for the communication to retrieve dynamic data, this project focuses its communication on the Marvel API[1].

The basic requirements were:

- **Retrieve dynamic data** (text and images) from a remote source (Flickr API). i.e. Retrieve a list of the latest photos uploaded to Flickr.
- **Present the dynamic data** in a table, grid or similar using a UITableView or UICollectionView. i.e. Display thumbnails of latest photos on Flickr in a list with captions.
- **Provide navigation** between two or more different types of scenes driven by dynamic data. i.e. Tapping on a photo loads a high-resolution version with additional details.

Additionally, extended requirements were encouraged to be implemented.

Please note that this project was implemented and tested for an iPhone X.

[1] https://developer.marvel.com

Chapter 2

Overall Design

The app is split into 4 main tabs:

- Favorites
- Marvel Database (default)
- Search
- Quiz

Those sections will be discussed and shown in this chapter, while also explaining the overall design of the iOS application with some screenshots.

2.1 Favorites

The Fav Tab consists of a ViewController that has three sections (Comics, Characters and Creators), and the corresponding CollectionViewController that will be presented if the user clicks one of the sections. To make it slightly more exciting, **LTMorphingLabels** were used to make a transition from the german words(Zeitschriften, Figuren, Authoren), which are shown in the beginning, to the english ones after three seconds. This effect is only shown when the Controller is first loaded.

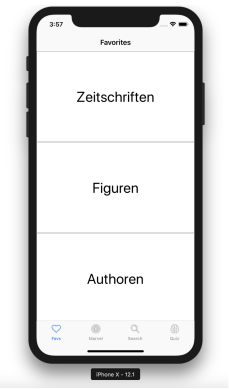

Figure 2.1: Fav Tab in the beginning

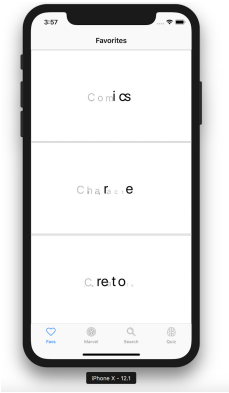

Figure 2.2: Fav Tab while morphing

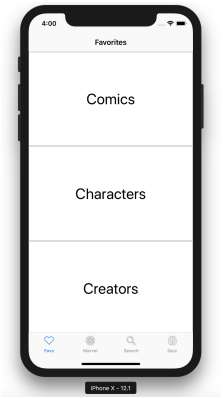

Figure 2.3: Fav Tab when done

Like previously mentioned, when the user selects one of the sections, the stored data will be displayed in a CollectionView. If the user hasn't marked any objects yet as his/her favorites, an alertmessage will pop up to make the user aware in how to store favorites, shown in figure 2.4. If the user has previously selected favorites, they will be shown as in figure 2.5.

At the top of the CollectionViewController there is a BarButtonItem, which looks like a trashcan. If the user wishes to delete all of his/her fav in this category, he/she can do so by clicking this button. As a safety-net, an alert message will be shown, asking if that is really what the users wants. This alert can be seen in figure 2.6.

Figure 2.4: Fav Tab alert message when nothing is stored

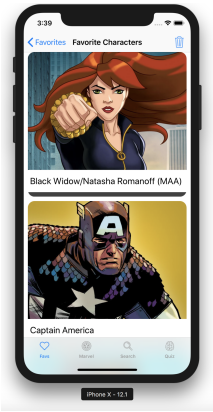

Figure 2.5: Fav Tab showing fav characters

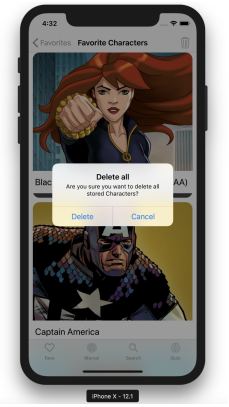

Figure 2.6: Fav Tab alert message when use wants to delete all

2.2 Marvel Data

The Marvel Data Tab consists of a ViewController that has a segment control with three sections (Comics, Characters and Creators). Depending on the section selected by the user, the controller hides or shows one of three CollectionViewController, which are embeded in a container.

The data is retrieved from the Marvel API and therefore sorted by name (A-Z).

The CollectionViewControllers allow the user continuous scrolling, therefore it will load more data into the view when the user reaches a certain point in the list. At the top of the controller, there is an BarButtonItem, which looks like a refresh arrow. When the user wants to go back to the top of the list and refresh the collected data, he/she can use this button to do so.

Figures 2.7, 2.8 and 2.9 show those scenes.

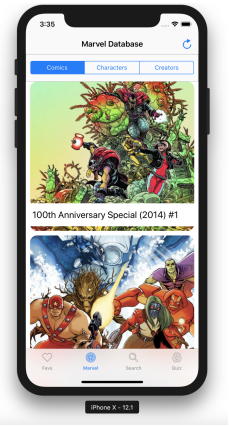

Figure 2.7: Marvel Tab comics

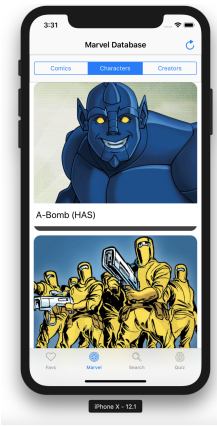

Figure 2.8: Marvel Tab characters

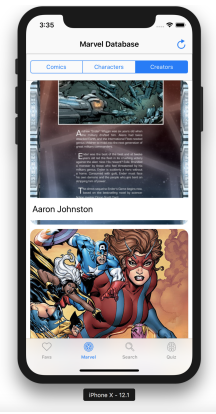

Figure 2.9: Marvel Tab creators

2.2.1 DetailView

If the user wants to see more information about the object, he/she can do that by simply selecting the cell. This will trigger the DetailViewController, which shows a bigger picture of the object, the full name, and some additional information, i.e. comic appearances for characters, involvements for creators, or characters and creators for comics. This is the default behaviour of all cells displayed (in Fav-, MarvelData- and Search-Tab).

As seen in figure 2.10 there is a big picture shown of the object - in this case a character named Captain America. This header is a "Sticky Header", which grows in size if one pulls it down.
Next to the name, there is a small heart-shaped button. This button can be used to select the character as a favorite by simply clicking it. The heart will then fill red, which indicates that it was successfully stored. If the user then switches back to the Fav Tab, he/she will see the character in the list.
At the top of the controller, there are two BarButtonItems. The very right one is to allow the user to share the picture on social media (depending on what is installed on the phone) or mail/messenger. The left button is for saving the picture to the user's library, if the permission was previously granted.

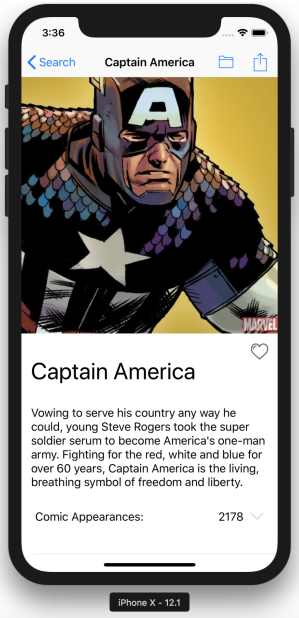

Figure 2.10: DetailView for character example - Captain America

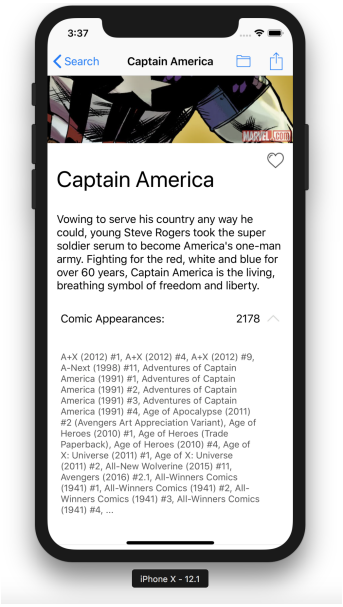

Figure 2.11: Extended DetailView for character example - Captain America

After the description (if there is any stored), more information will be provided in the TableView that is located beneath the descriptionLabel, depending on what was stored the API's database. In the case of Captain America, it will show how many comics he has appeared in. If the user wants to see some examples of his comics, they can press the small arrow next to the number, which will then expand the cell and show the first 20 comics. An expanded cell is shown in figure 2.11.

The cell can be shortened again by using the same arrow (which is now pointing in the other direction).

Unfortunately, the Marvel API still has some bugs, which include that a search request with "(" or ")" characters is always going to fail. As every comic has its year in brackets in its title, it is not possible to request comics by name and display them again. Same goes for the comics that the creators were involved in.
However, at the DetailView of a comic it is possible to show the mentioned creators (as they don't have brackets in their title). The user can just press the "Show more" button at the bottom to go to a new CollectionView which holds only the creators involved in the specific comic. This can be seen in figures 2.12, 2.13 and 2.14.

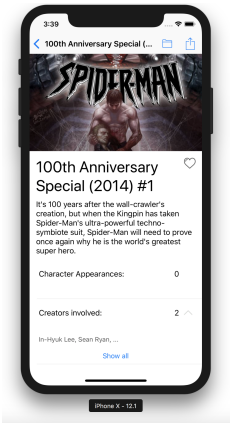

Figure 2.12: Extended DetailView for comic with creators

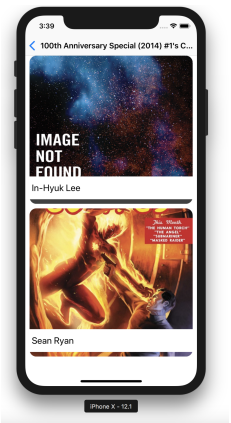

Figure 2.13: DetailCollectionView for creators involved in comic

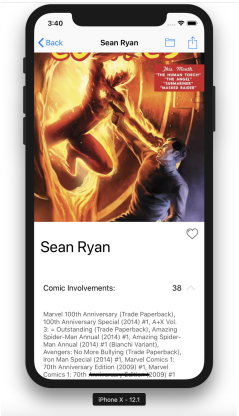

Figure 2.14: Extended DetailView for creator

2.3 Search

The Search Tab allows the user to search through the whole database for a phrase defined by him/her. Per default, the database will look for characters, but if the user wants to change this (to comics or creators), he/she can do so by pressing the NavigationBarTitle, which is a **BTNavigationDropdownMenu** and will then show all filters. Afterwards the user can write a phrase or name into the searchBar and press "Search" on the keyboard or press anywhere on the screen to trigger the search request. If the database doesn't hold the requested object, an alert will tell the user to define the search parameter differently. If the database returns something, the results will be shown in a CollectionView.
Mentioned use-cases can be seen in these figures:

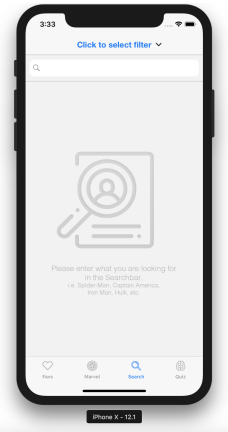

Figure 2.15: Search Tab empty

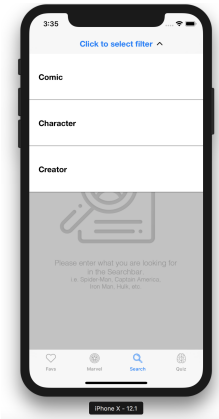

Figure 2.16: Search Tab picker

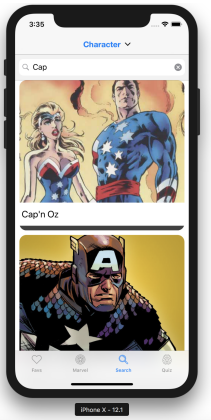

Figure 2.17: Search Tab results for "Cap"

2.4 Quiz

The last scene - the Quiz Tab - can be seen in figure 2.18. This is a quick game to allow the user to test his/her knowledge about the characters. Out of all characters that were already fetched, the system randomly selects one to be recognized, and three different names. The controller keeps track of the score (+100 for each correct answer). If the user selects a wrong answer, the game will stop. If the user has beat the highscore, an alert message will congratulate him/her, whereas it wishes better luck next time if the user lost.

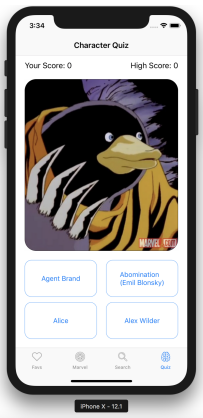

Figure 2.18: Quiz Tab start of game

Chapter 3

Requirements

This chapter explains which basic and extended requirements were met by the project's result and how they were implemented.

3.1 Basic Requirements

3.1.1 Retrieve dynamic data

To communicate with and retrieve data from the **MARVEL API**[1], two helper classes were implemented: *DataHandler* and *Decoder*. The DataHandler prepares the URL (which also includes an api key, md5-hash[2] and timestamp), and starts the request to the API. If it was successful, the DataHandler will give the results to the Decoder, which - as the name indicates - decodes the information (json) to objects.
The DataHandler holds different functions, that can either request only the simple data (name, id, thumbnail) and save those in one of three arrays (*comics*, *characters* or *creators*), or request detailed information about a specific object (by id or name) to be shown in the DetailView, or displayed in the Search- or FavTab.
If it requests simple data to store in the arrays, it will filter all results based on if they hold a picture or not. Therefore, objects that don't hold their own images, aren't displayed in the MarvelData Tab. To allow this filtering, a dictionary *alreadyRequestedAmount* was implemented that holds the amount of objects that were already requested (as this varies a lot from the amount that was stored in the arrays).

3.1.2 Present dynamic data

An UICollectionView is used to show the objects, whereas a small version of a UITableView is used in the DetailsView for the expandable cells about appearances / involvements / creators / characters.

[1] See https://developer.marvel.com

[2] Function to generate a MD5 hash was found on http://iosdeveloperzone.com/2014/10/03/using-commoncrypto-in-swift

3.1.3 Provide navigation

Following Storyboard in figure 3.1 can explain the navigation withing the app.

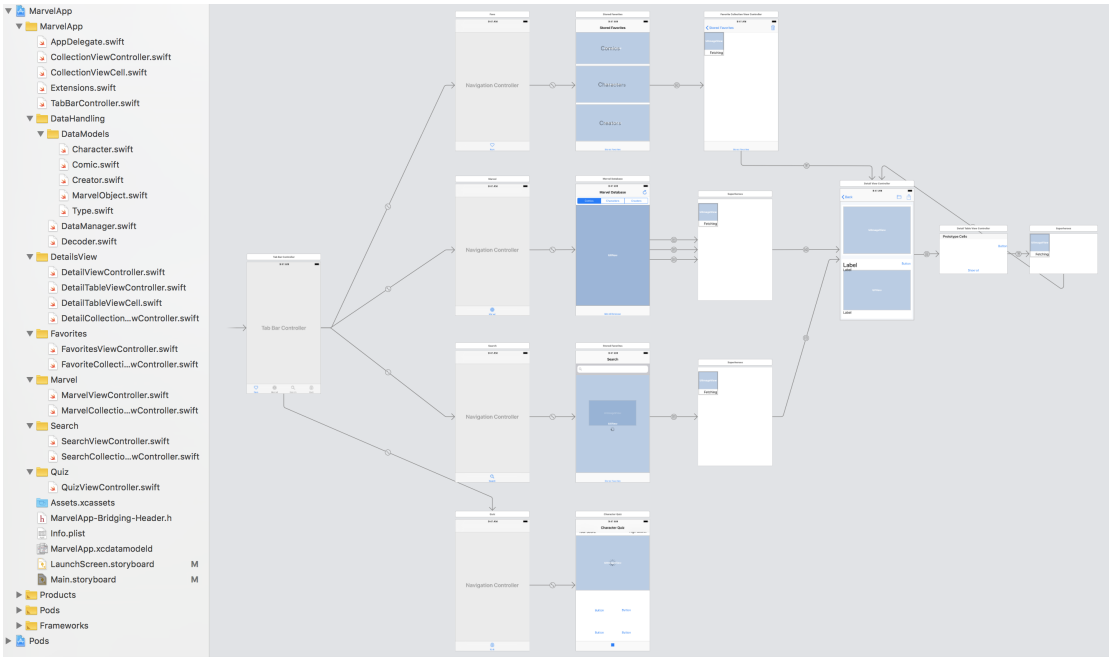

Figure 3.1: Storyboard

The app has four tabs, which are handled by the *TabBarController*. Each tab has its own NavigationController to secure that the user can go back easily (or have a title for the Quiztab). The first tab is the FavTab, which connects to the *FavoritesViewController*. When the user selects one of its sections, it will present the *FavoritesCollectionViewController*. The *FavoritesCollectionViewController* inherits from *CollectionViewController*, therefore acts the same as its parent.

When a cell gets selected, the *CollectionViewController* will then redirect to the *DetailViewController*. The DetailViewController has a container that embeds the *DetailTableViewController*. If the object is a comic and the comic has some creators, a "show more" button will allow the user to jump to the *DetailCollectionViewController* where the creators, which were involved in creating the comic, will be shown. From this *DetailCollectionViewController* it is of course possible to see the details again, therefore the circle closes and one gets redirected to the respective *DetailViewController* of the creator.

The second and default tab is the MarvelDataTab, which connects to the *MarvelViewController*. This VC holds three containers that connect to the *MarvelCollectionViewController*. The *MarvelCollectionViewController* inherits from *CollectionViewController*, therefore acts the same as its parent.

The third tab is the SearchTab, which connects to the *SearchViewController*. This VC

holds a container that connects to the *SearchCollectionViewController*. The *SearchCollectionViewController* inherits from *CollectionViewController*, therefore acts the same as its parent.

The last tab is the QuizTab, which connects to the *QuizViewController*. This VC doesn't present any other controller, except for an alertVC when the player loses.

3.2 Extended Requirements

3.2.1 Notify users of long operations

For every connection operation in the searchTab, there is an UIActivityIndicatorView on the controller which shows the user that something is going on in the background. The indicator will be started by calling *.startAnimating()* when the search is triggered either in the *searchBarSearchButtonClicked(searchBar:)* or *touchesEnded(touches:, with event:)*. A timer then checks every second if a result has been found. If this is true, the indicator will be stopped with *.stopAnimating()* and be hidden by setting *.isHidden = true*.
The same is happening in the QuizTab, while the system waits to get at least 10 characters to choose from. The imageView and buttons are hidden, while the indicator in the backround is animating. Again, a timer is scheduled to check every second if there are >= 10 objects already in the characters array from the datamanager. If it is true, the indicator will be stopped and the imageView and buttons will be shown again by setting their attribute *.isHidden = false*.
Additionally, every *CollectionViewCell* has an idicator as well, that will be shown and animated until the cell has fetched the image.
Furthermore, with the help of the pod **Reachability**, that is made use of in the *AppDelegate*, the app checks for an active internet connection and triggers an alert message if there is no internet available.

Figure 3.2: Loading Cell

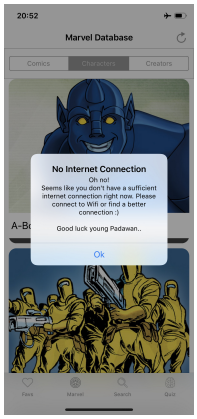

Figure 3.3: Internet Connection was lost

Figure 3.4: Loading Quiz

3.2.2 Progressive loading

To allow continuous scrolling, progressive loading was implemented for the MarvelDataTab. Which means that when the user scrolls to the end of the CollectionView, new data will be requested from the API.

As it could happen that the user scrolls too fast for the system to catch up (since the API is sometimes quite slow), *collectionView(collectionView:numberOfItemsInSection)* returns the total amount of available data entries. Therefore, the user will always see cells, even if they are still loading.

collectionView(collectionView:prefetchItemsAt:) gets called when the user scrolls to the bottom, and it then checks with the self-made helper method *isLoadingCell(indexPath:)* if the upcoming cell already has a respective object in the dataManager. If not, it will request 20 more to be loaded. If the request is successful, the collectionView will reload its entries with

```
1 DispatchQueue.main.async {
2   self.collectionView.reloadData()
3 }
```

Listing 3.1: reload data in collectionview

3.2.3 Social Media

To allow the user to share pictures of the marvelObjects, the DetailView has a BarButtonItem for this, which triggers the sharing feature. The code to do this is shown below:

```
1 @IBAction func share() {
2   let share: [Any] = [imageView.image!, "Uhhhh.. look whom I've found! Check \(
      marvelObject.name) out at MarvelApp!"]
3
4   let activityViewController = UIActivityViewController(activityItems: share,
      applicationActivities: nil)
5   activityViewController.popoverPresentationController?.sourceView = self.view
6   self.present(activityViewController, animated: true, completion: nil)
7 }
```

Listing 3.2: IBAction share()

In the variable *share: [Any]* we store two things: the image we want to share, and the message that should for example be displayed in the body of an email. Then we create a new ActivityViewController and initialize it with our share variable. Afterwards we tell the activityVC over which VC it should present itself, and then tell the DetailViewController to present the activityCV.

3.2.4 Using additional frameworks

As the Marvel API doesn't provide location based information, the app has no point using i.e. MapKit. But instead, the app has installed four pods that were used:

- SnapKit[3] for the layout and alignment
- Reachability[4] for checking the internet connection
- BTNavigationDropdownMenu[5] to have a picker in the NavigationTitle
- LTMorphingLabel[6] for the morphing effect in the Fav Tab

Therefore, the Podfile looks like:

```
1 # Uncomment the next line to define a global platform for your project
2 platform :ios, '9.0'
3
4 target 'MarvelApp' do
5 # Comment the next line if you're not using Swift and don't want to use dynamic
      frameworks
6 use_frameworks!
7
8 # Pods for MarvelApp
9 pod 'SnapKit', '~> 4.0.0'
10 pod 'ReachabilitySwift'
11 pod 'BTNavigationDropdownMenu'
12 pod 'LTMorphingLabel'
13
14 end
```

Listing 3.3: Podfile

3.2.5 Allow users to navigate through additional dynamic datasets

In the DetailView of the comic-objects it is possible to trigger another CollectionView if the comic has creators. The comic-object retrieves additional information (like description and the names of its creators) when it switches to the DetailView. If the creators should be displayed, a new request will be sent, searching for the names of the creators. The results will be shown in a new CollectionView - see figure 2.13 in earlier chapter.

3.2.6 Add meaningful animations

Like mentioned in an earlier chapter about the design, the FavViewController uses LTMorphingLabel to have a transition from german to english words to make the view more appealing. It is triggered easily with a timer that changes the text after a certrain amount of time.

Another animation is on the same screen: When the user opens the FavCollectionView of a category that has nothing stored, an alert will pop up and explain how to add favorites. When the user dismisses the alert, the controller will be popped, so that the user finds him-/herself back at the parentView. Why was this implemented? It doesn't really make sense to stay at a view where there is nothing shown.

[3] https://github.com/SnapKit/SnapKit
[4] https://cocoapods.org/pods/ReachabilitySwift
[5] https://github.com/PhamBaTho/BTNavigationDropdownMenu
[6] https://github.com/lexrus/LTMorphingLabel

3.2.7 Add search functionality

See figure 2.15 for more visual input. The search functionality was implemented in its own tab - the SearchTab. When the user has written something in the searchBar and pressed "search" on the keyboard or clicked anywhere else (and the input is not empty), the SearchVC will tell the DataManager to request all object of the same name and the *selectedType* and will display it in a SearchCollectionVC.

As mentioned earlier, it also implements a BTNavigationDropdownMenu, with which one can select another category to search.

```
let items = ["Comic", "Character", "Creator"]
let menuView = BTNavigationDropdownMenu(navigationController: self.
    navigationController,
    containerView: self.navigationController!.view,
    title: BTTitle.title("Click to select filter"),
    items: items)

menuView.didSelectItemAtIndexHandler = { (indexPath: Int) -> () in
    switch indexPath {
      case 0: // comics
        self.selectedType = .comics
      case 1: // characters
        self.selectedType = .characters
      default: // creators
        self.selectedType = .creators
    }
    self.collectionViewController.objectsToShow = nil
    self.collectionViewContainer.isHidden = true
}
...
self.navigationItem.titleView = menuView
```

Listing 3.4: How to setup the BTNavigationDropdownMenu

At the beginning we put the strings into an array, which later will be the different options to pick from, and initialize the BTNavigationDropdownMenu menuView with our navigationController, its view, the title to be displayed in the beginning, and our array of options.
Then we tell the menuView what to do when a new item was selected - in our case we want to change the *selectedType* and set everything to default again.
In the end we have to set the titleView of the current navigationItem to our menuView.

3.2.8 Support for fav pics

The heart icon in the DetailVC allows the user to save favorites, which will then be displayed in the corresponding FavTab. To do so, the *UserDefaults* are used. To save new values, the system calls *UserDefaults.standard.array(forKey: marvelObject.type.rawValue + "FavId") as? [Int]* to retrieve the currently stored values as an Integer array. Then it adds the new id at the end and overrides the old value by calling *UserDefaults.standard.set(favs, forKey: marvelObject.type.rawValue + "FavId")*, where fav is the array with the appended new value.
Same happens when a fav gets deleted, only that the system calls *let newFavs =*

favs.filter {$0 != marvelObject.id} to delete (all) values in there with the same id as the current object, before it gets stored into the UserDefaults again.

3.2.9 Refreshing in Favs

If the user is in the DetailView of an object while being in the FavTab, and then deciding to delete this object from his/her favs by selecting the heart icon, the CollectionView will refresh itself when coming back, so that the deleted object is no longer shown. This was implemented by using *requestData(forType: favoriteType)* in *viewWillAppear()*.

3.3 Own features

3.3.1 Saving pictures to phone

Saving pictures directly to the phone library is also implemented. The call *UIImageWriteToSavedPhotosAlbum(image, self, #selector(image(image:didFinishSavingWithError:contextInfo:)), nil)* calls the method *image(...)*, which handles the results (whether it was stored or not) with an alertmessage. It will only store the picture if the user grants the app permission to write into the library though!

3.3.2 Labels in DetailsView fit length

With setting *.numberOfLines = 0* and *.lineBreakMode = .byWordWrapping*, the labels will have multiple lines and change their size depending on the input.

3.3.3 Scroll to top

By using *.setContentOffset(.zero, animated: true)* on the correct collectionView, the collectionView will scroll up to the top again, no matter how far down the user has scrolled before.

3.3.4 Sticky Header

With a scrollView, which holds two different containers (one for the imageView, and one for the rest of the screen), and a lot of constraints, it was possible to implement an image-Header that expands when dragged down, but then jumping back to its original position. Three of the most important lines here were *scrollView.contentInsetAdjustmentBehavior = .never*, *imageView.contentMode = .scaleAspectFill* and *imageView.clipsToBounds = true*.
The remaining constraints were implemented by using SnapKit. A simple SnapKit use can be seen in this code snippet:

```
1 imageContainer.snp.makeConstraints { make in
2   make.top.equalTo(scrollView)
3   make.left.right.equalTo(view)
4   make.height.equalTo(imageContainer.snp.width)
5 }
```

Listing 3.5: Constraint set with SnapKit

This code sets the constraint for the imageContainer. The top constraint is the same as the top constraint for the scrollView. The left and right constraints equal those from view. And the height is defined the same as its width (making it a square).

3.3.5 Expandable TableViewCells in DetailViewController

This was quite tricky. It requires a tableView to be implemented in the contentContainer in the DetailViewController. To be able to have this (but not ruining the existing sticky header in the process), an extension *optimalHeight* for the UILabel was added to calculate how much space the title and description label need, so that under it the tableView can start.
When the user clicks the arrow to expand the cell, the new cell height has to be calculated depending on how much text must be displayed. To calculate the height of the informationLabel, an extension *height(constraintedWidth:font:)* was added to String, that then could be used to find out the height of the cell by adding the informationLabelHeight, titleLabelHeight and showMoreButtonHeight.

3.3.6 Delete all favs in category

By overwriting the current value in the UserDefaults for the specified category to an empty Integer array, users can delete all stored objects in a category.

3.3.7 Quiz

An own tab was introduced to allow users to test their knowledge about all marvel characters. The highScore is stored in the UserDefaults and will only be overwritten if the user scores higher. With *Int.random(in: 0 ..< dataManager.characters.count)* an index for the to-be-guessed character will be randomly chosen, the same will be used to find three more (different) characters to show their names as other options.

3.3.8 Alert when favs empty

If the selected category in the FavTab is empty, an alertmessage will pop up, explaining how to store favorites and to come back later when more are added. When the user dismisses the alert, the FavoriteCollectionViewController will be popped, so that the user is back at the FavoriteViewController.

3.3.9 Alert when search wasn't successful

If the search request wasn't successful because the system couldn't find the parameter in the database, an alert message will pop up, explaining the situation and giving examples of successful requests.

3.3.10 Different API

A different API was chosen for this project, which required very intense dataHandling. The main implementation for dataHandling is happening in the classes *DataHandler* and *Decoder*.

Chapter 4

Acknowledgements

4.1 SnapKit

SnapKit was used to be able to add constraints programmatically.
https://github.com/SnapKit/SnapKit

4.2 Reachability

Reachabilty was used to check for active internet connection.
https://cocoapods.org/pods/ReachabilitySwift

4.3 BTNavigationDropdownMenu

BTNavigationDropdownMenu was used to have a picker for the categories in SearchTab.
https://github.com/PhamBaTho/BTNavigationDropdownMenu

4.4 LTMorphingLabel

LTMorphingLabel was used to have an animation in the FavTab.
https://github.com/lexrus/LTMorphingLabel

4.5 Stackoverflow

Of course, some bugs were not fixed on my own, but with the help of the community. Looking into Stackoverflow from time to time if stuck on a problem, is quite helpful. Therefore, big shoutout to the community - thanks!
https://stackoverflow.com

4.6 iosDeveloperZone

The function to generate a MD5 hash was found on iosDeveloperZone
http://iosdeveloperzone.com/2014/10/03/using-commoncrypto-in-swift

Chapter 5

Conclusion

Overall, I have had a lot of fun developing this application. I have learned a lot in respect to API requests and datahandling, and had time to have a look at different pods. Especially working with SnapKit was new for me, and I am very glad I decided to include it in this project.