COMP39X

2019/20

A Seat Assignment System

Student Name:	Xueyang Liu
Student ID:	201376191
Supervisor Name:	Prof Frank Wolter

DEPARTMENT OF

COMPUTER SCIENCE

University of Liverpool

Liverpool L69 3BX

Dedicated to Liming and Ying, my parents.

Acknowledgements

I'd like to thank my parents, Liming and Ying, for their support throughout my studies. Thanks also to my girlfriend, Pengmin, for comforting me when I was upset and providing professional artistical opinions on the UI design of my software.

I dedicated this to all who helped me throughout my project, my project supervisor and my roommates.

COMP39X

2019/20

A Seat Assignment System

DEPARTMENT OF

COMPUTER SCIENCE

University of Liverpool

Liverpool L69 3BX

Contents

Abstract

This project implemented an iOS cinema app using SwiftUI. The finished app allowed the user to view movie information, select seats and book tickets. A database used to store all movies, tickets and users' information was built on a cloud server which was managed using Realm, a newly emerged database management tool. The cloud server also enabled data sync among multiple devices. This project built a qualified app that could be used for real commercial use, although the size may limit.

1 Introduction

1.1 Project Description

Generally, the project was to design and implement a cinema seat assignment iOS application, allowing customers to search for movies, book tickets and choose the seats' positions. The special point was that users can set their seat preferences and the app will recommend suitable choices. The reason why the idea of this system was raised is that not all the customers could afford the time to go to the cinema in advance to buy a ticket. Also, sometimes the queueing lines in cinema were too long and communicating with the staff to choose the preferred seat could take much time. This application, with all information and every choice the user could make clearly shown on the screen, would simplify all these complex and time-consuming processes into several clicks. It could therefore be used by the customers as a remote and more time-efficient way to book tickets and choose seats.

1.2 Background

1.2.1 Odeon app

Due to the fact that the seat assignment system on iOS was not as many as on webpages, there were few examples to review. In this part, the student analysed the Odeon application(*ODEON on the App Store*, 2020), which could be found in app store on iOS devices. The analysis mainly focused on the basic working principles of the seat assignment and the whole procedure from searching films to finally confirm seats. Although it was a widely used application in the UK and Ireland, the rating was not decent, only 2.4 out of 5, and most reviews regarded it as "an OK app but could be better". The analysis would include some advantages that deserved learning from as well as some disadvantages that needed improving.

- Advantages
    1. The Odeon application allowed every user to search films, watch trailers and read reviews, not including a log-in process unless the user wants to book a ticket. This was considered to have reduced a lot of redundant data. Some users might just want to have a look at recent films and not even want to go to the cinema. There was no need to get their data stored.
    2. The application had very clear steps and simple user interface, so it would be time saving for those in a rush.
    3. Meanwhile, for those who wanted to spend some more time on this app, it provided the "favourite" function, the rating and commenting function, and so on.
    4. The seat selection process was reasonable. It successfully meets the need of the most, but it can still be improved.
- Disadvantages

1. There was a subtle rule for the seat selection in Odeon app. The rule was that a single seat cannot be left. For example, for the four seats near the aisle, the app would forbid some choices to make sure the one seat near the aisle will not be left alone. This could be good to the cinema, because this ensured efficient use of all the seats. But it took part of the customers' freedom of choice.
2. The app did not include a notification function. It would be better if the app could push some movies that users might be interested in.
3. When buying tickets for two, only one membership card could be used. This might be something not elegant in the way they organized their database, that they could verify and stored the information of only one card for one account.

After experiencing the Odeon app, an enterprise-level application, the student was familiar with what a decent seat assignment app looked like, and so the implementation method behind it.

1.2.2 Realm and its cloud server

There were many ways to manage the generated data on iOS devices, such as SQLite, CoreData and so on. Due to the fact that the complexity of implementing SQLite on iOS devices was relatively high, and CoreData did not allow data sync among different devices, the student finally chose to use Realm(*Realm*, 2020). Realm is an open source object database management system designed for mobile app development and is now trusted by lots of companies and organisations. It is developer friendly and enable real-time sync for data, which is exactly what the project needs.

By using Realm, a database server, all data in the database could be stored in the

server and fetched to the mobile when needed.

The cloud server of Realm was one of its paid service and the student made use of its one-month trial to develop the app. The cloud server enabled the sync of data among different devices.

1.2.3 SwiftUI

SwiftUI (*SwiftUI*, 2020) was a relatively new way to build user interfaces across all Apple platforms. It was introduced to the public on WWDC 2019 and used declarative Swift syntax, which made the coding process easier and more natural.

1.3 Aims & Objectives

1.3.1 Aims

1) This project aimed to create an application that allows the customers to view movie information, order tickets and select their seats on their apple devices.
2) This app would recommend seat choices according to the users' seat preferences to save their time.
3) The managers of the cinema would be able to customize the seat plans and change movie information.
4) After the basic functions was completed, hopefully a more advanced algorithm would replace the preference settings and be used to generate recommendations of seats.

1.3.2 Objectives

1) The application should provide different user interfaces to customers and managers with different options.
2) The application should be able to run on different devices and the

databases should be consistent by transmitting data via the Internet.

3) The application should be able to transmit data fast and properly to and from the database.
4) The UI design should be as simple as possible, as well as pragmatic.

2 Design

2.1 Functional design

The functions of this app should be similar to those popular apps on the market, such as the Odeon app analyzed above.

The app needed to provide the following functions:

For all users:

- User log in and sign up
- View movies

Login needed:

- Add movies to favorite list
- Set seat preferences
- Order tickets and select seats
- Cancel tickets
- Log out

2.2 Database design

A database is required in this app to store user and movie information. By using the Realm database, the cloud server was able to store objects which were defined by the developer in the app. The design of the database would be illustrated in this section, discussing the objects required and the relationships between them.

First of all, for each user, a User object was needed to store his/her login information, the username and password, and also his/her tickets information, favorite movies and seat preferences. Then, for each movie, a Movie object was required, with the attributes including movie name and other information about the movie. Also, each seat with the screen it belonged to and its specific showtime was stored as an object. The class diagram (Figure 1) following showed the very first design of the classes needed and relationship

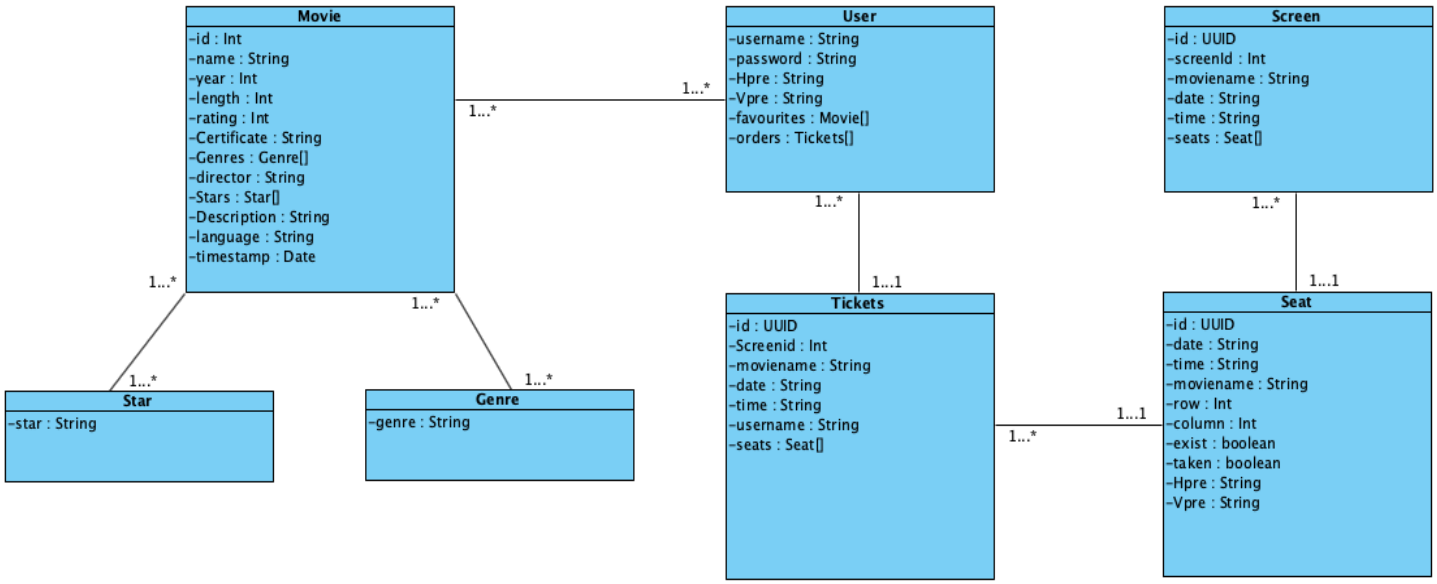

Figure 1: Class diagram and database structure

between them.

The Realm database was special because it could directly store objects, so the figure was not only a class diagram but also a database structure, with the first attributes under each object being the primary key.

2.3 User case Diagram

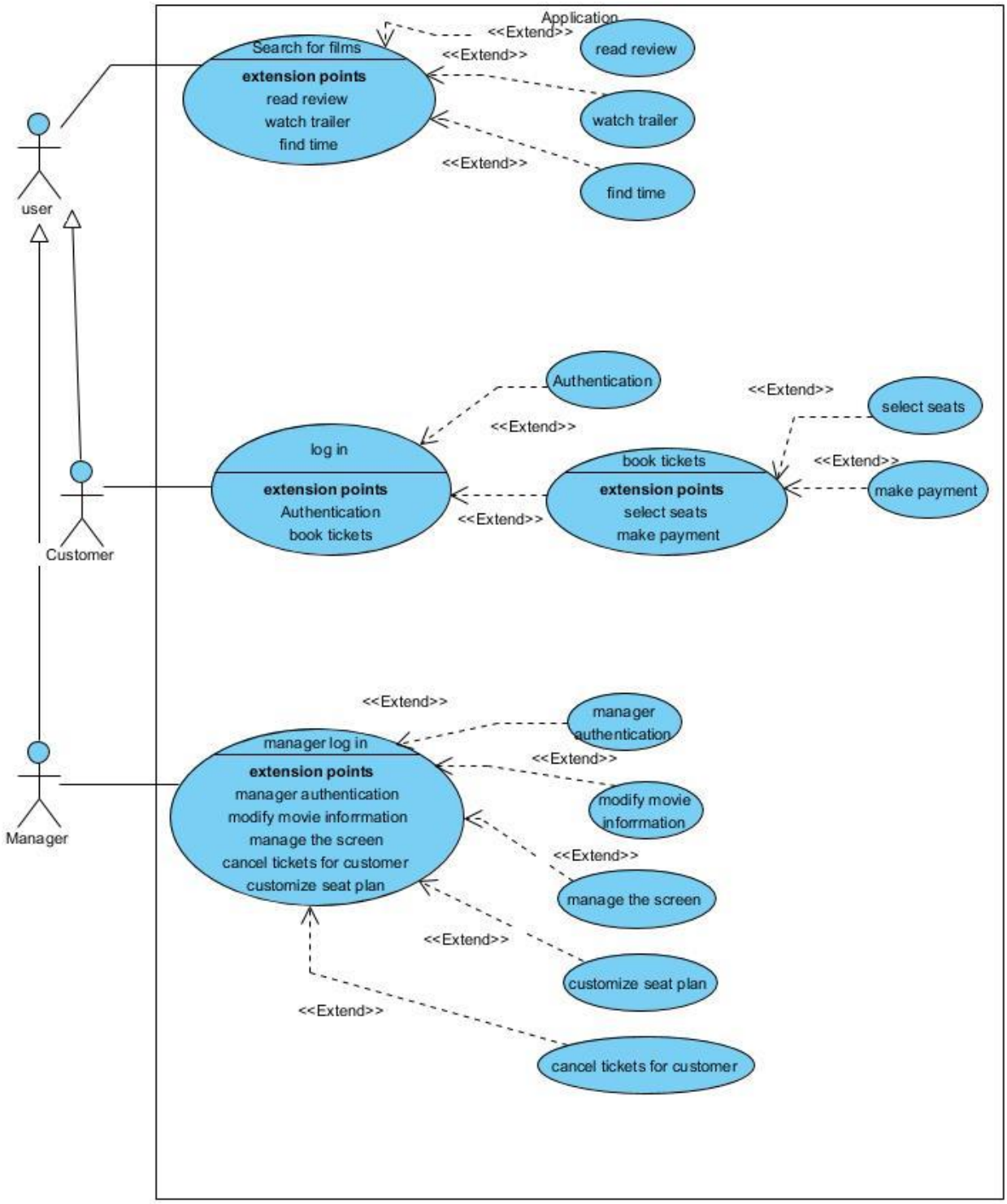

Figure 2: User case diagram

3 Implementation

3.1 Development model

In this project, the desired development model would be the prototype model. Since the application would include a lot of interaction with the end users, this development model would ensure that the end users constantly work with the application and provide feedbacks to improve it.

3.2 Software & Hardware Resources

The This app was developed on the students own MacBook Pro, using XCode, which is a free software. An open source database management tool, Realm, was considered necessary to implement the app. Although Realm itself was free to use, its cloud server, which was also needed in this project to enable the sync of data among different devices, was a paid service. The developer used the one-month free trial to develop the app. If the app was for commercial use, a company license needed to be purchased. The test and evaluation of the application will be conducted through the simulators of XCode and the student's own iPhone 8. Additionally, the student will be using Microsoft office software such as Words for writing reports, Project for time planning and work management, PowerPoint for presentation, which were all provided by the University of Liverpool.

3.3 Realm Cloud Server

The Realm cloud server was where the database was built. The app would connect to the database and log in as an admin user. The point was that the connection was not a one-time event. That is to say the app could maintain connected to the database and

querying for data as long as it did not log out actively.

```
1/let creds = SyncCredentials.usernamePassword(username: Constants.admin, 2/password:
Constants.password, register: false)
3/                SyncUser.logIn(with: creds, server: Constants.AUTH_URL, 4/onCompletion:
{ (user, err) in
5/                    if let error = err {
6/                        self.errormsg = "Failed to connect the database"
7/                        self.error = true
8/                        print("Failed to connect the database: \(error)")
9/                        return
10/                   }
11/                   print("Connected to DB!")
12/                   let config = SyncUser.current?.configuration(realmURL: Constants.
13/MOVIES_URL, fullSynchronization: true)
14/                     self.realm = try! Realm(configuration: config!)
```

The code above shows the log-in process to the Realm cloud server. In line 1 and 2, the credentials were encapsulated into the object SyncCredentials. From line 3 to 10 is the log-in process with possible error handling. The Constants.AUTH_URL in line 3 is the cloud server's authentication server's address. From line 12 to line 14 is the process of configuring the realm object at the mobile by referring to the cloud server. Upon the realm object is configured successfully, it will keep "full synchronization" with the server. By using the following syntax, objects stored in the realm database can be fetched.
Take Movie object as an example,

```
let movieResult = self.realm?.objects(Movie.self).filter("moviename='Joker'")
```

the return of calling objects to realm is basically a list of Movie objects stored in the realm database and satisfied the NSPredicate. NSPredicate can be used to further filter the result and get more accurate results. In the code above, the movieResult is still a list, only the list gets just one Movie object "Joker".

3.4 SwiftUI Syntax

In this section, several concepts in SwiftUI would be introduced so that the code snippet

in the next “Code Implementation” section could be easier to understand.

- Stack Views

  There were HStack{}, VStack{} and ZStack{} in SwiftUI. H stood for “Horizontal”, V stood for “Vertical” and Z for “Z-axis”. The content in the curly brackets would be arranged according to them. See Figure 3 as an example.

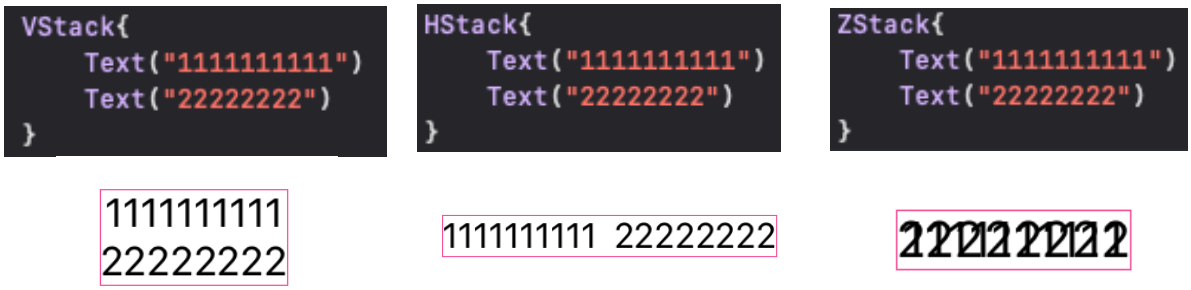

Figure 3: VStack, HStack and ZStack

  In VStack, the first item came to the top and the following would be arranged down to the top. Similarly, in HStack, the first item came to the left side and in ZStack, the first item was overlayed by the following ones.

- GeometryReader

  GeometryReader was used to obtain the screen’s size information and the obtained value could help adjust the size of view no matter on what device the app ran. For example, in Figure 4, the yellow background was adjusted to half the width of the screen with the help of GeometryReader.

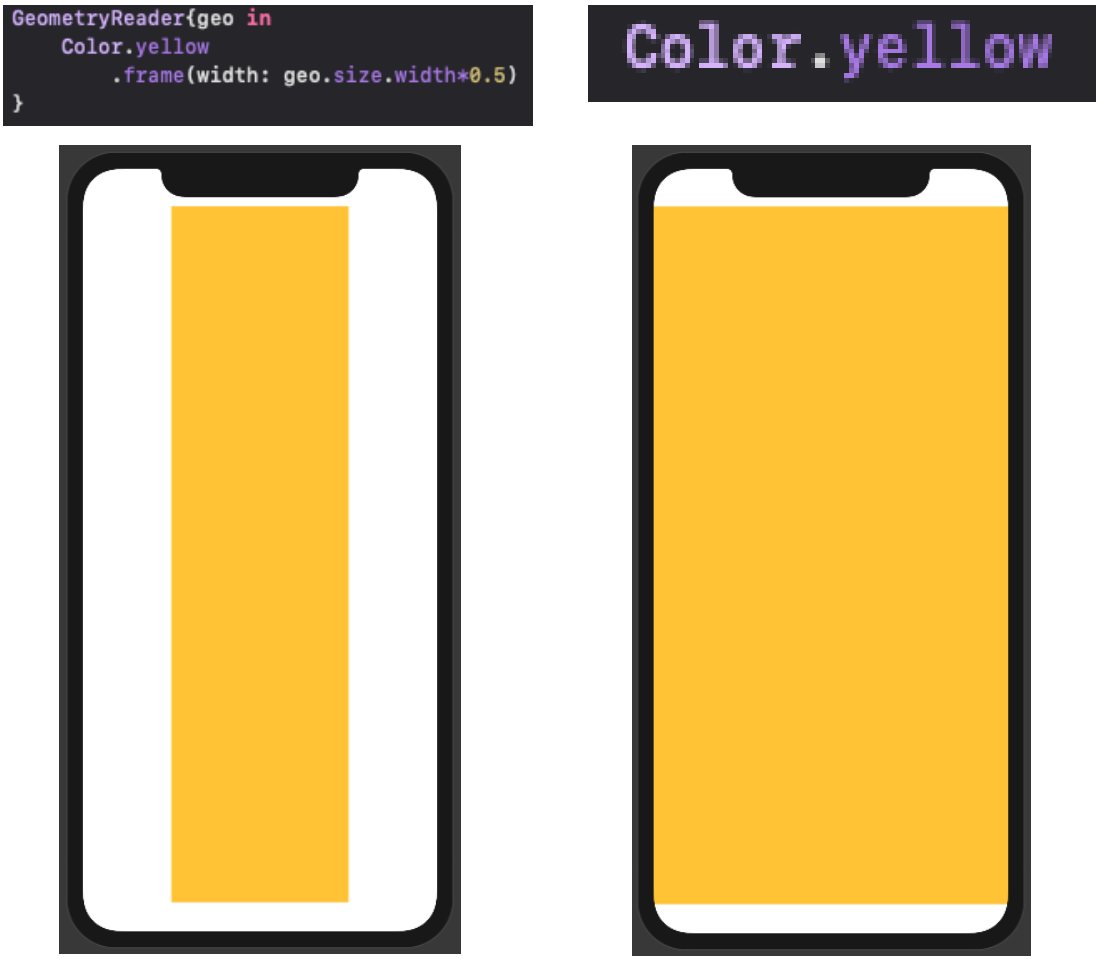

Figure 4: GeometryReader

- SF Symbols

  SF Symbols (*SF Symbols*, 2020) was a set of over 1,500 consistent, highly configurable symbols which was provided by Apple and free to use when developing apps. A program called SF Symbols could be downloaded from Apple Developer website and was basically a search program. As Figure 5 shown below, the results would show the image and its name in system for reference.

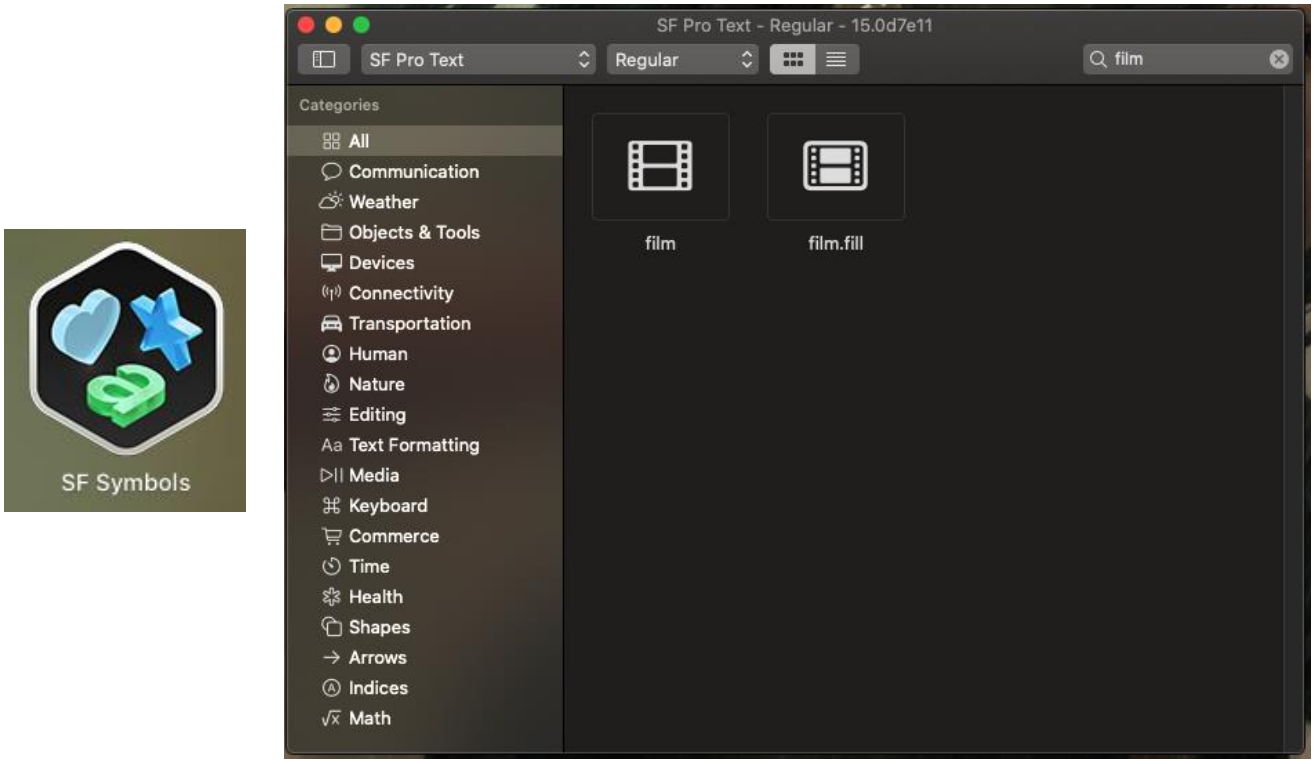

Figure 5: SF Symbols Application

In development process, the Image could be used as the following Figure 6 shows, with only one line of code.

```
Image(systemName: "film")
```

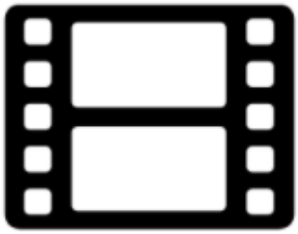

Figure 6: System image "film"

3.5 Code Implementation

This section would explain the code of this app in detail about the implementation of each page and function, using the order in which a regular user would use the app. All figures in this section would be snapshots of the XCode iPhone 11 simulator, which were exactly same with the running results on a real iPhone.

3.5.1 Loading View

When a user open this app, the first view displayed would be the loading view, as shown in Figure 7. This view would overlay all other views before the app was successfully connected to the database. The implementation of this view is shown in the code snippet below (Figure 8).

```
ZStack{
    BlurView()
    VStack{
        Indicator()
        Text("Loading...")
            .foregroundColor(Color.white)
            .padding(.top,8)
    }
}
.frame(width:110,height:110)
.cornerRadius(10)
}
```

Figure 8: Loading View (code)

Figure 7:Loading View

The implementation was to overlay a loading indicator and a text saying “Loading…” on a blur view. The blur view would blur the current page so that the user could not see behind and have to wait the app loading. It was used while the app opened and was connecting to the database, which would normally take a period varying between 3 - 5 seconds.

3.5.2 Connect to Realm Cloud Server

When the main page appeared, a function would be called and the app would connect to the database. The code of the function is shown in Figure 9.

```
func connectDB(){
    if !self.stayConnectSuccess{
        for u in SyncUser.all{
            u.value.logOut()
        }
        let creds = SyncCredentials.usernamePassword(username: Constants.admin, password: Constants.password, register: false)
        SyncUser.logIn(with: creds, server: Constants.AUTH_URL, onCompletion: { (user, err) in
            if let error = err {
                self.errormsg = "Failed to connect the database"
                self.error = true
                print("Failed to connect the database: \(error)")
                return
            }
            print("Connected to DB!")
            let config = SyncUser.current?.configuration(realmURL: Constants.MOVIES_URL, fullSynchronization: true)
            self.realm = try! Realm(configuration: config!)
            while self.realm == nil{}
            self.informConnectSuccess = true
            self.stayConnectSuccess = true
        })
    }
}
```

Figure 9: Log in to cloud server

First of all, the connectDB() function would only be executed when the Boolean variable stayConnectSuccess was false. Then, to avoid multiple login conflict, for all users had logged in to the Realm cloud server, they must logged out. SyncUser was a class defined by the RealmSwift module and was used to control all Realm user issues. After all users logged out, the function used the pre-set credentials to log into the Realm cloud server and catch the error if there was one. Then the realm object was configured for further usage. At the end of this piece of code, after the realm object was successfully built, the stayConnectSuccess was turned to true and would maintain true in the rest of time. Otherwise, every time the user switch to the main page, this function would be executed and waste some time because the realm cloud log in process normally consumed $3 - 5$ seconds. The informConnectSuccess Boolean was bound to an "Login Success" alert, which would show up upon it became true.

In the rest time of using the app, the communication with the database would be conducted via the Realm object: self.realm, and the average waiting time would be significantly shorter than the SyncUser login part, which was hard to notice.

3.5.3 Transition Between Tabs

As a single view app, the transition between pages was essential. In this app, there were three main pages: Movie view, Favourites view and Member view. Users were expected to frequently switch between pages while using the app. The SwiftUI provided a suitable view called TabView for this situation. The tabs would stay at the bottom and above it was the page which the user viewed. The code is shown in the code snippet below(Figure 10).

```
TabView{
    AllMoviesView()
        .tabItem{
            Image(systemName: "film")
            Text("Movies")
        }
    FavouritesView()
        .tabItem{
            Image(systemName: "star")
            Text("FAVOURITES")
        }
    MemberLoginView()
        .tabItem{
            Image(systemName: "person")
            Text("MEMBER")
        }
}
.background(Color.black)
.accentColor(Color.yellow)
```

Figure 11: TabView (code)

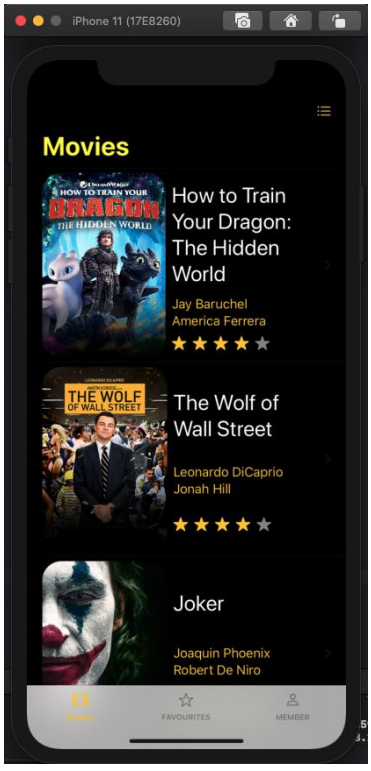

Figure 10: TabView, allMoviesView

In Figure 11, the current tab was AllMoviesView() and the page showed all movies in the cinema's database. The "tabItem" in the source code was corresponding to the tab button on the simulator. Each "tabItem" consisted of an icon and a text. The icons were suitable images chosen from SF Symbols and whose accent colour were changed to yellow to match the "cinema" theme.

3.5.4 Showing all movies

In the AllMoviesView(), a list of all movies in the database was displayed. In this page, an array was used to store Movie object obtained from the Realm server. The array was initialized as an empty one and filled with movies after the app connected to the database.

```
@State private var movies:[Movie] = []
```

```
func getAllMovies(){
    while self.realm == nil {}
    let movieResult = self.realm?.objects(Movie.self)
    print(self.realm?.objects(Movie.self).count ?? 0)
    for elem in movieResult!{
        self.movies.append(elem)
    }
}
```

Figure 12: Acquire all movies

When all movies were fetched to the array, the information was ready and the problem was how to display them. The following code snippet and visual result (Figure 12) would help to explain this.

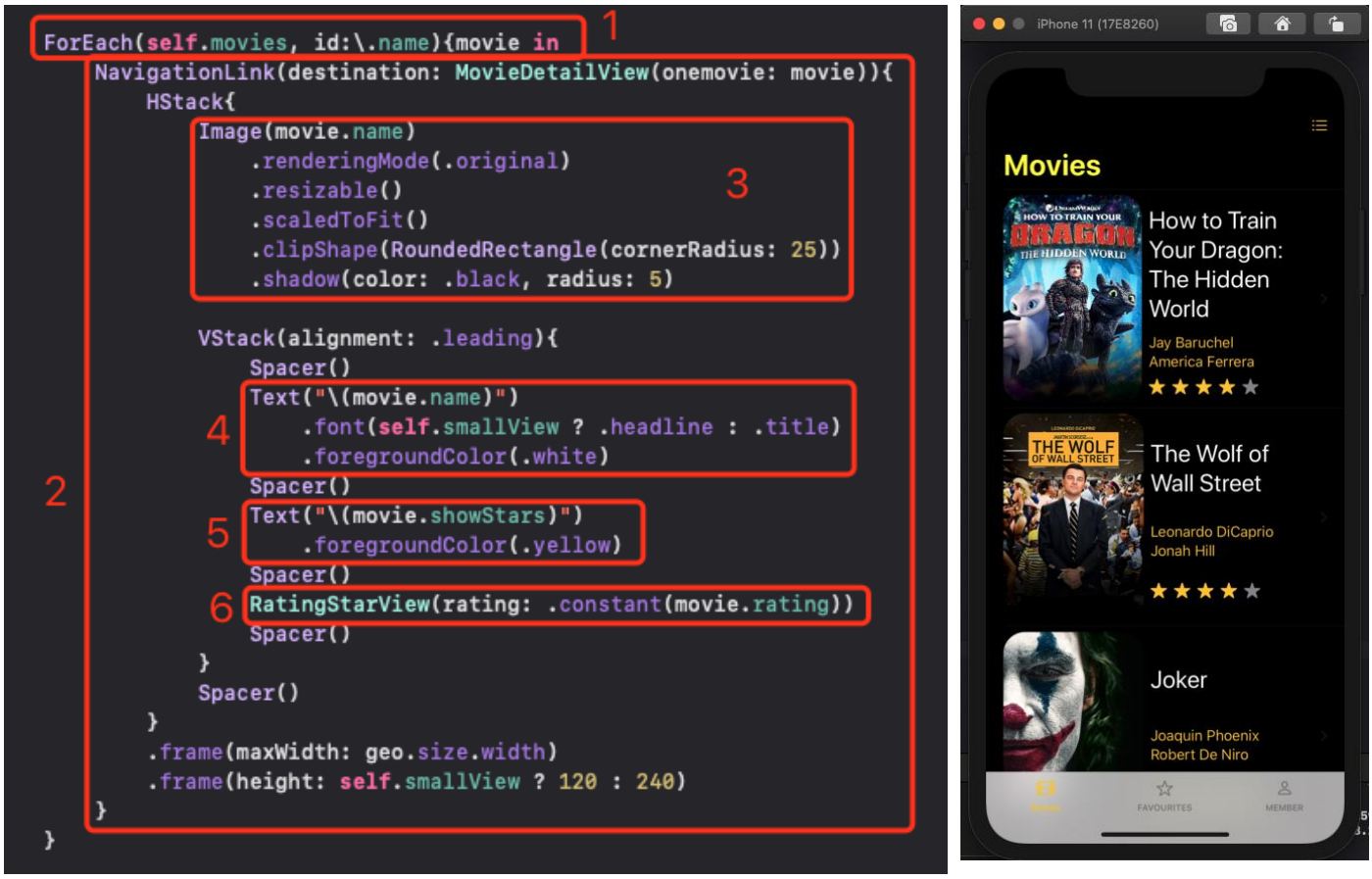

Figure 13: Arrangement of movie info (code and visual effect)

The first block in the code was a travel process with the movie's name as identifier and each movie was used in the following block 2, which meant for each movie, the code in block 2 was executed and corresponding UI was generated. Since tapping on each movie would lead the user to its detail page, the developer used NavigationLink with the destination MovieDetailView and the movie object passed in. The NavigationLink stood for the function of each each area, and the appearance would be decided by the contents inside the curly brackets behind it. Inside them was a HStack containing a VStack, showing a poster via the code in block 3, the movie name via the code in block 4, the stars' names in block 5 and a rating view in block 6.

The RatingStarview was a separate view used to visualize the movie's rating. It was implemented by the following code.

```
HStack{
    ForEach(1..<6, id:\.self){ num in
        Image(systemName: "star.fill")
            .foregroundColor(num>self.rating ? Color.gray : Color.yellow)
    }
```

Figure 14: RatingStarView (code)

The theory was simple: horizontally show five stars and for the rating n between 1 and 5, the first n stars would be yellow and the others be gray.

These components combined together to form one NavigationLink for a movie. All movies' NavigationLinks were put into a list and the user could scroll to view.

A useful function was added to this page while the beta version was tested by the volunteers, some of which complained that the movies' posters were set to big and they were only able to glance two and a half movies at one time without scrolling. As a result, a button for the thumbnail mode was added to the trailing top corner of the page. The visual effect and code implementation are shown below in Figure 15 and 16.

```
.navigationBarItems(trailing:
    Button(action: {
        self.smallView.toggle()
    }){
        Image(systemName: self.smallView ? "square.grid.2x2":"list.bullet")
            .padding([.top,.leading,.bottom],3)
})
```

Figure 15: Thumbnail Button

```
ForEach(self.movies, id:\.name){movie in
    NavigationLink(destination: MovieDetailView(onemovie: movie)){
        HStack{
            Image(movie.name)
                .renderingMode(.original)
                .resizable()
                .scaledToFit()
                .clipShape(RoundedRectangle(cornerRadius: 25))
                .shadow(color: .black, radius: 5)

            VStack(alignment: .leading){
                Spacer()
                Text("\(movie.name)")
                    .font(self.smallView ? .headline : .title)
                    .foregroundColor(.white)
                Spacer()
                Text("\(movie.showStars)")
                    .foregroundColor(.yellow)
                Spacer()
                RatingStarView(rating: .constant(movie.rating))
                Spacer()
            }
            Spacer()
        }
        .frame(maxWidth: geo.size.width)
        .frame(height: self.smallView ? 120 : 240)
    }
}
```

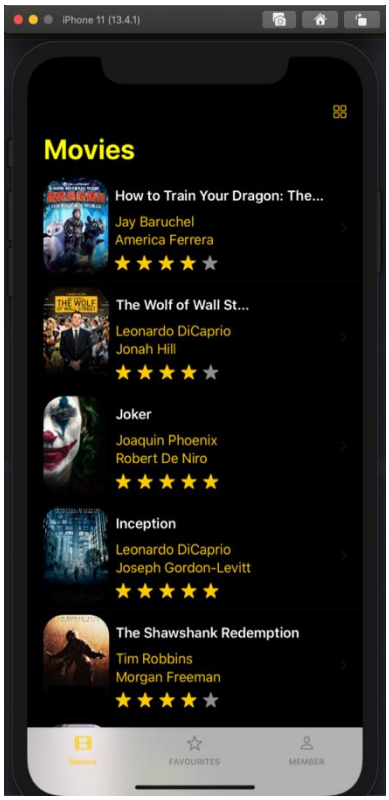

Figure 16: Implementation of changable size

The frame of each NavigationLink was set to a width same as the screen and a height decided by the Boolean self.smallView, which would toggle every time the user tapped the button. Since the poster was resizable and scaled to fit, it would automatically change with the frame. Also, the font of the movie name was set to vary according to the Boolean value. In this way, a thumbnail view was implemented.

3.5.5 User Login View

Continued to the AllMoviesView, it was ought to be each movie's detail page to be discuss, but due to the fact that functions in that page require user logging in, the user login view would be introduced first. When the user tapped the "MEMBER" button at the trailing bottom corner of the screen, he/she would enter the user login view, which looked like Figure 17.

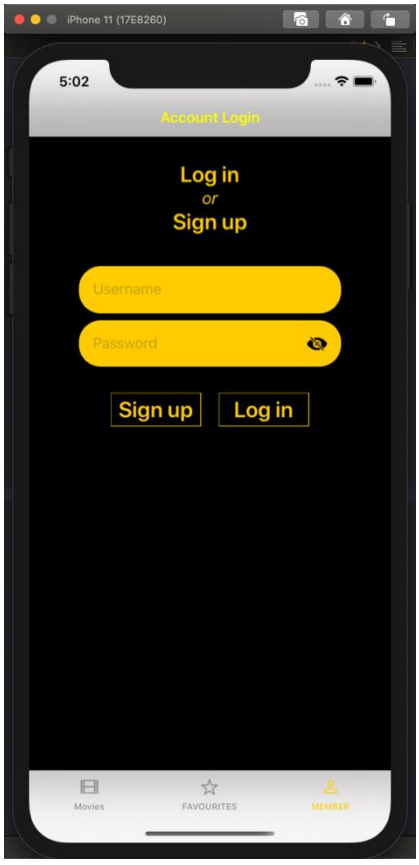

Figure 17: Signup or Login View

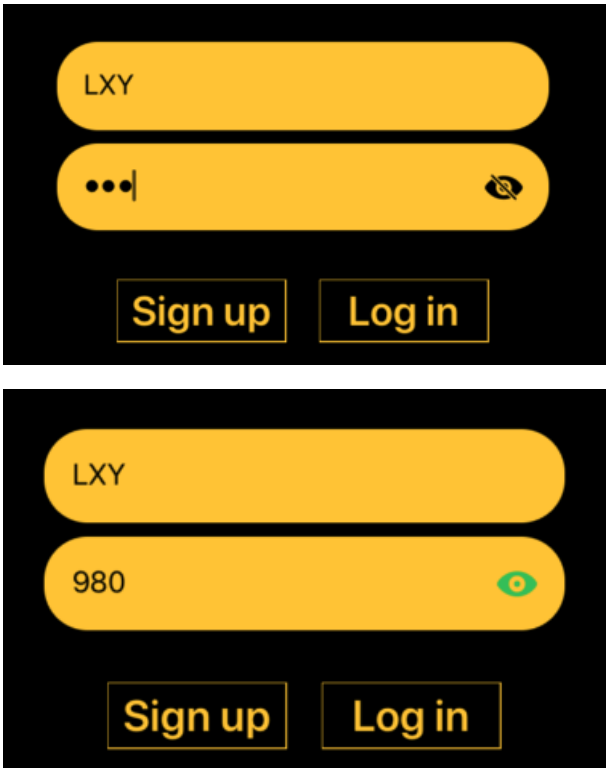

Figure 18: Show or Hide the password

Figure 18 shows a fucntion that was added during the bata version test. Some volunteers hoped to have a toggle to show or hide the password. This function was implemented by showing TextFiled or SecureField according to the toggle status and overlaying an eye image on the field, with its color changing with the toggle as well.

```
ZStack(alignment: .trailing){
        if self.show {
            TextField("Password", text: self.$password)
                .accentColor(.black)
                .padding()
                .frame(width: 300)
                .background(Color.yellow)
                .cornerRadius(25)
        }else{
            SecureField("Password", text: self.$password)
                .accentColor(.black)
                .padding()
                .frame(width: 300)
                .background(Color.yellow)
                .cornerRadius(25)
        }
    Button(action:{
        self.show.toggle()
    }){
        Image(systemName: self.show ? "eye.fill" : "eye.slash.fill")
            .padding()
            .foregroundColor((self.show==true) ? Color.green : Color.black)
            .offset(x:0)
    }
```

Figure 19: Implementation of hiding the password

SecureField was a subclass of TextField, with all chars in it displayed as dots and inaccessible. But they both could be used to take the user's input through the binding string $password and stored in self.password. Similarly, the username was taken though an TextField and stored in self.username. Upon acquiring the username and password, the system would be able to perform sign up or log in function by comparing the two information with records in the database.

The two functions were similar so they were written as one function with the Boolean "register" to show the difference.

```
func signup(){
    verify(username: self.username, password: self.password, register: true)
}
func login(){
    verify(username: self.username, password: self.password, register: false)
}
```

Figure 20: user signup and login

The verify function included several judgements about the username and password, as well as the Boolean "register". The log in failed and sign up failed cases would be:

1) One or both of the username and password were empty.

2) When logging in, the username was not in the database.

3) When logging in, the password and username did not match.

4) When signing up, the username was already taken.

The source code was too long and here only the code including operations to the database would be discussed.

```
try! self.realm!.write{
    self.realm!.add(newuser)
}
self.showAlert = true
self.alerttitle = "Success!"
self.alertmsg =
"""
You have registered successfully!
Now log in!
"""
print("Successfully registered!")
```

Figure 21:Sign up successfully

When successfully signing up, the app would write the new user record into the database, and show a message to inform the user.

```
print("User login succeeded!")
self.showAlert  = true
self.alerttitle = "Success!"
self.alertmsg = "Logged in successfully!"
self.loginuser = existuser
self.logsuccess = true
regularuser.username = username
regularuser.password = password
regularuser.Hpre = existuser!.Hpre
regularuser.Vpre = existuser!.Vpre
print(regularuser.Hpre)
print(regularuser.Vpre)
```

Figure 22: Login successfully

When succeeding in logging in, the app would also show a message to inform the user, and there would be some additional operations.

First, the user object would be passed to self.loginuser for later use in this page. And the Boolean logsuccess would be toggled to true. This value related to what to be shown in the member login page, because the basic structure of this page was as shown in the pseudocode below (the real code was too long to be shown).

```
if !self.logsuccess{
    memberLoginView()
}else{
        LoggedInView()
}
```

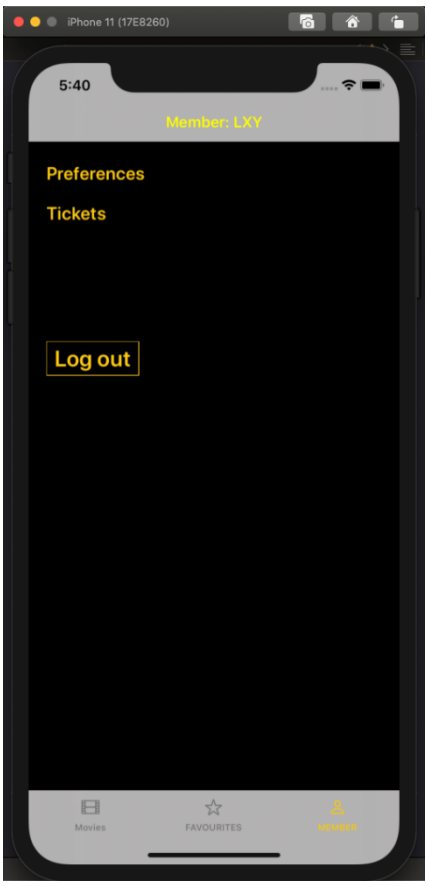

Figure 23: Logged in page

Upon the success of logging in, the page would show the user's personal page, as Figure 23. The user could change his/her seat preferences and view/cancel his/her booked tickets through the two NavigationLink and these two pages would be discussed later.

The "Log Out" button would execute the following code when tapped.

```
func logOut(){
    self.logsuccess = false
    regularuser.username = ""
    regularuser.password = ""
    regularuser.Hpre = ""
    regularuser.Vpre = ""
}
```

Figure 24: Log out (code)

The logsuccess was toggled to false and therefore the page would show the login view

again. And the rest of code about regularuser might be familiar. They also occurred in the previous login success part of code. The regularuser is an EnvironmentObject which used to share one particular object among many swift files. Going back to the TabView page, a regularuser instance was initialized and passed into all pages.

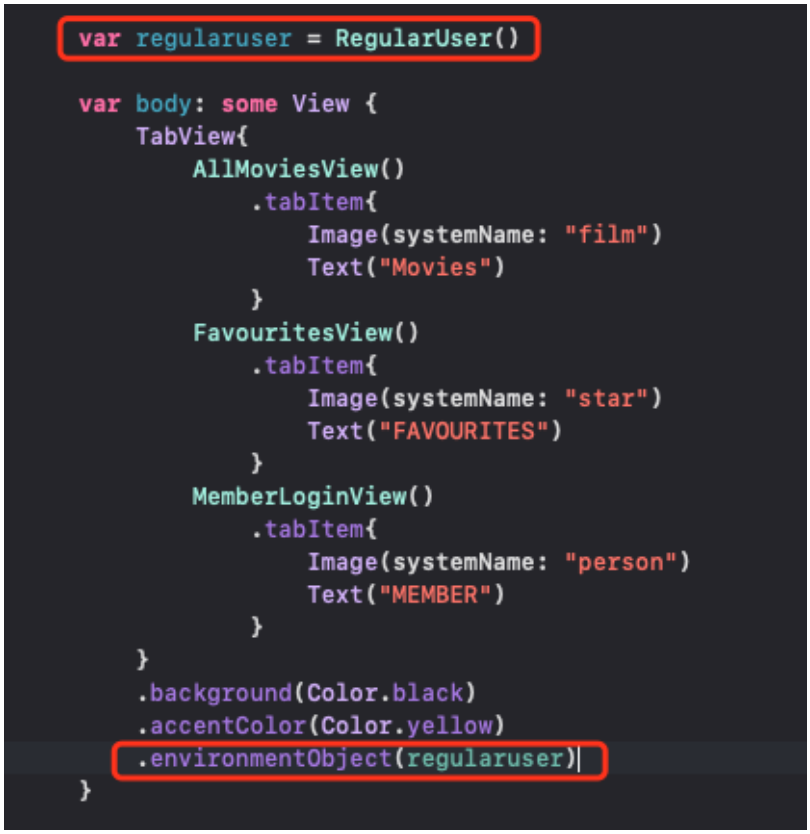

Figure 25: regularuser object passed into every page

This object can be accessible and amend in different pages and keep synced. Therefore, once the user logged in successfully, the app would extract its information and stored it into the regularuser object, sharing among pages, until the user tapped “Log Out” and its information would be erased from the object. The previous mentioned functions which need user logging in in the movie detail view would be enabled once they detected the username and password of the regularuser object are not empty.

3.5.6 Movie Detail View

This view intended to show the user all information about the movie in detail, with an

entrance to the booking page. The implementation of this page was basically how to arrange all information using different Stacks and two buttons, “favorite” button and “book now” button. This section would focus on the two buttons since the page arrangement using Stacks was similar to previous pages. This was a ScrollView so it took two pictures (Figure 26) to show the content arrangement.

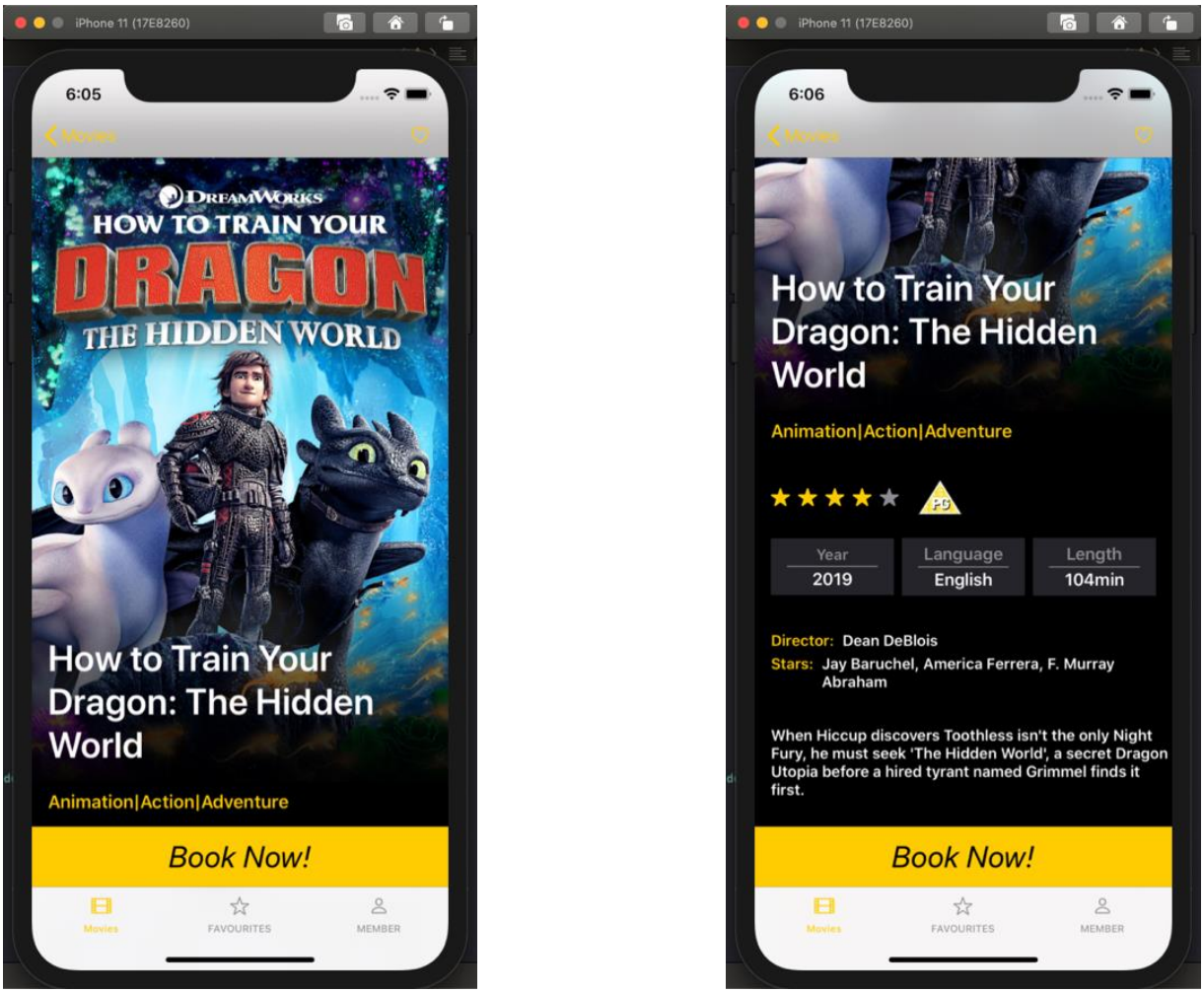

Figure 26: Movie detail view

It is noticeable that the “Book Now!” button constantly floats at the bottom of the view and above the tabs. This was implemented by overlaying the button on the ScrollView using ZStack. The small heart on the title bar was for the favorites function. Only logged in users could use these two functions, so a judging process was demanded.

About the favorite function, when this movie detail view showed up, an examination was

already done through the following function (Figure 27).

```
func load(){
    let config = SyncUser.current?.configuration(realmURL: Constants.MOVIES_URL, fullSynchronization: true)
    self.realm = try! Realm(configuration: config!)
    if self.regularuser.username != ""{
        let favs = self.realm?.objects(User.self).filter("username = '\(self.regularuser.username)'").first?.favourites
        for movie in favs!{
            if movie.name == self.movie.name{
                self.isFav = true
            }
        }
    }
}
```

Figure 27: Examination upon loading

The function acquired the database first and then saw if the user had logged in by checking the regularuser. If no user logged in, it would do nothing, which was reasonable because even if not logging in, all users of this app should be able to see the detail of the movie, as long as they did not tap the two buttons. Otherwise, for the logged in user, the app would examine his/her favorites to see if the movie of this page is in his/her favorites list and if true, it would toggle the Boolean “isFav” to true, which was bound to the heart image. Turning “isFav” to true would change the heart into a filled one.

Figure 28: heart and heart.fill

Also, as a logged in user, the user would be able to toggle the heart status to add the movie into his/her favorites or remove it from them. This function was implemented as below.

```
func favouriteToggle(){
    if self.regularuser.username == ""{
        self.NoUser = true
    }else if self.isFav{
            let favlist = self.realm?.objects(User.self).filter("username = '\(self.regularuser.username)'").first?.favourites
            var newFavlist:[Movie] = []
            for movie in favlist!{
                if movie.name != self.movie.name{
                    newFavlist.append(movie)
                }
            }
            try! self.realm!.write{
                while favlist!.count>0{
                    favlist?.remove(at: 0)
                }
                for movie in newFavlist{
                    favlist?.append(movie)
                }
            }
            self.isFav = false
        }else{
            let favlist = self.realm?.objects(User.self).filter("username = '\(self.regularuser.username)'").first?.favourites
            let AddMovie = self.realm?.objects(Movie.self).filter("name = '\(self.movie.name)'").first
            try! self.realm!.write{
                favlist?.append(AddMovie!)
            }
            self.isFav = true
        }
    }
```

Figure 29: How to toggle favorite

The first thing to do when the user tapped the heart button was to check if he/she was a logged in user. If true, an alert bound to the Boolean "NoUser" would pop up and inform the user to log in first and dismiss the current page, returning to AllMoviesView(). Otherwise, there were two circumstances.

1) Due to the fact that the Realm did not allow the deletion of one specific object in the list. Under the first one, which the movie was already one of the user's favorites and he/she wanted to remove it from the list, the implemented method was to get the list of favorite Movie objects from the database first. Then, for each movie, if it is not the one shown in the current detail page, append it to a new Movie array. Afterwards, remove all movies in the favorite list. At last, add movies in the new Movie array one by one into the favorite list and turn the Boolean "isFav" to false to show an empty heart.
2) If the movie shown in the page was not among the user's favorites, fetch the favorite list and the movie object, then add it into the list and turn the Boolean "isFav" to

true to make the heart full.

3.5.7 Select Date and Screen

The second function in this page was the "Book Now!" button, which was a NavigationLink that would take the user to the page to choose tickets number and the showtime. The judging process of user was conducted in the destination page. The page would be like Figure 30 (left) if there was no user information in the regularuser object.

Figure 30: No user, Select time and Not on

However, if the regularuser contained a username, which denoted a valid user, the page would correctly show up and let user choose tickets number, the date and screen to watch the movie. Moreover, if the movie was not on any more and there were no available showtimes, the page could also display a message for that.

The implementation of this page mainly related to the Screen object in the database. The pseudocode was written below to describe the logic:

If possibleScreen[] is not empty{

SelectView()

}else{

Text("The movie is currently not on!")

}.alert(shown if no user and dismiss the current page)

When the page was loaded, the possible screens were obtained by querying the database (Figure 31).

```
func load(){
    let config = SyncUser.current?.configuration(realmURL: Constants.MOVIES_URL, fullSynchronization: true)
    self.realm = try! Realm(configuration: config!)
    while self.realm == nil{}
    let getScreens = self.realm?.objects(Screen.self).filter("moviename = '\(self.moviename)'").sorted(by:["date","time"])
    self.allPossible = []
    for elem in getScreens!{
        self.allPossible.append(elem)
    }
    self.NoUser = (self.regularuser.username == "")
    self.loadCompleted = true
}
```

Figure 31: Acquiring all possible screens

The query obtained all Screen objects with the specific movie name and sorted by date and time. Then these Screens were appended to a Screen array, allPossible. But till now, the app could not check the available seat number in each screen. For example, in the situation that the user needed 5 tickets but the screen only had 4 available seats, it was unreasonable to let user choose that screen. As shown in Figure 32, a stepper was on the top and ask the user to choose a ticket number, the number there was bound to a variable, seatsCount.

```
Stepper("How many tickets?            \(self.seatsCount)", value: self.$seatsCount, in: 1...48)
    .foregroundColor(Color.yellow)
    .accentColor(Color.white)
    .background(Color.secondary)
```

Figure 32: Stepper

The seat number was chosen between 1 and 48 because the cinema in this app could have most 48 seats. According to the seat number chosen here, the screens were filtered and only those with sufficient seats could stay on the list. This was implemented through two functions in Figure and Figure .

```
func countAvailableSeats(screen:Screen)->Int{
    var count = 0
    for seat in screen.seats{
        if seat.exist == true && seat.taken == false{
            count+=1
        }
    }
    return count
}
```

```
func selectAvailable(NumNeeded: Int)->[Screen]{
    var result:[Screen] = []
    for screen in self.allPossible{
        if countAvailableSeats(screen:screen)>(NumNeeded-1){
            result.append(screen)
        }
    }
    return result
}
```

Figure 33:countAvailableSeats() and selectAvailable()

The countAvailableSeats() function takes in a specific Screen object and returns the count of available seats. The selectAvailable() functions takes in the needed seat number filter the allPossible Screen array return an array of suitable Screen objects.

```
ForEach(self.selectAvailable(NumNeeded: self.seatsCount),id:\.id){screen in
    NavigationLink(destination: SelectSeatView(screen: screen,seatNum: self.seatsCount)){
        HStack{
            Text("Screen \(screen.screenId)")
                .italic()
                .foregroundColor(Color.yellow)
            Spacer()
            Text(screen.date)
                .bold()
                .foregroundColor(Color.white)
            Text(screen.time)
                .bold()
                .foregroundColor(Color.white)
            Spacer()
        }
        .frame(maxWidth: geo.size.width)
    }
}
```

Figure 34: Screen wrapped in a NavigationLink

The suitable Screens were wrapped into NavigationLinks and then shown on the page for user to choose from. If any NavigationLink was tapped, it would take the user to the seat selection page.

3.5.8 Seat Selection view

The UI implementation of this page were not hard, similarly to the previous ones. But the data structure might be the most complex one in the whole app, due to a special characteristic of the Realm database. It was found that all objects in the app configured for the Realm database could not be amended outside the try! realm.write{} brackets, even though the developer was not meant to write it to the database. Therefore, the developer figured out a method to avoid this rule and the method was used once more in the ordered tickets view.

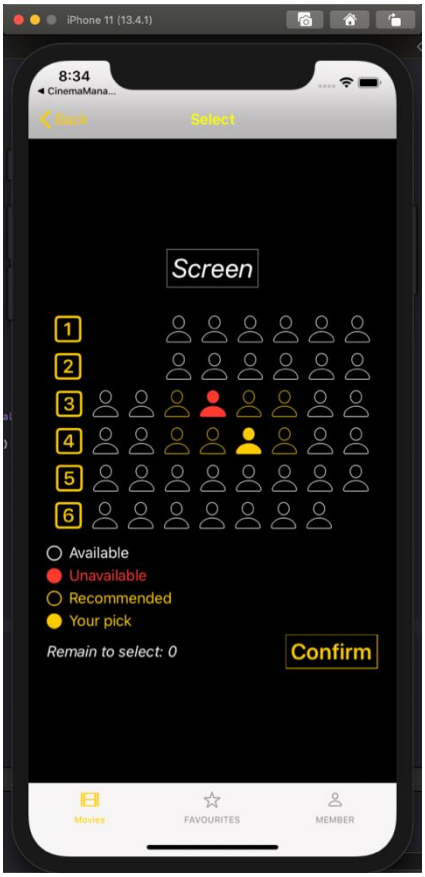

Figure 35: Select a seat

The desired UI of this page should look like Figure . When the user chose a seat, even though he/she had not confirmed yet, the seat should differ from other seats. To realize this function, every seat was designed to be a single view and would show different image in different statuses.

According to the seat plan, if there was no seat, like the top leading corner in Figure , there should not be seats. Available seats should be white, recommended seat should be yellow and empty, taken seat should be red and full, selected seat should be yellow and full.

The judgment was conducted through the following code:

```
SingleSeatView(id: seat.id, Hpre: seat.Hpre, Vpre: seat.Vpre, exist: seat.exist, taken: seat.taken,myseats: self.mySeat)
```

```
var body: some View {
    if !exist{
        return Image(systemName: "clear").resizable().scaledToFit().foregroundColor(.black).font(Font.title.weight(.ultraLight))
    }else if taken{
        if ifIn(id: self.id,idArr: myseats){
            return Image(systemName: "person.fill").resizable().scaledToFit().foregroundColor(.yellow).font(Font.title.weight(.ultraLight))
        }
        return Image(systemName: "person.fill").resizable().scaledToFit().foregroundColor(.red).font(Font.title.weight(.ultraLight))
    }else if Hpre == regularuser.Hpre && Vpre == regularuser.Vpre{
        return Image(systemName: "person").resizable().scaledToFit().foregroundColor(.yellow).font(Font.title.weight(.ultraLight))
    }else{
        return Image(systemName: "person").resizable().scaledToFit().foregroundColor(.white).font(Font.title.weight(.ultraLight))
    }
}
```

Figure 36: How to decide which seatView to show

The singleSeatView takes in the seat's UUID, the users preferences, the exist Boolean, the taken Boolean, and the selected seats' UUID array. And return the corresponding kind of single seat view. Each time the user tapped on a single seat, the seat's attributes needed to be changed. For example, the taken Boolean toggles, but before the user confirmed, he/she could cancel the selection and chose another seat. So, there was no reason to use try! realm.write{} to write any change which was not settled to the database. The class myFakeSeat was defined to solve this issue. A myFakeSeat object had the exactly same attributes with the real Seat object and a function called copytoAuth() which returns a real Seat object with all attributes the same. For the same need, the real Seat object had a function called copytoFake().

```
func copytoAuth()->Seat{
    let a = Seat(moviename: self.moviename, date: self.date, time: self.time, row: self.row, column: self.column, exist: self.exist,taken: self.taken)
    a.id = self.id
    return a
}
```

```
func copytoFake()->myFakeSeat{
    let a = myFakeSeat(moviename: self.moviename, date: self.date, time: self.time, row: self.row, column: self.column, exist: self.exist,taken: self.taken)
    a.id = self.id
    return a
}
```

Figure 37: "copy" function allowed easy conversion

With fake seats being used, the attributes could be amended and all the app needed to do was update the database when the user click the confirm button. Fake seats would be copytoAuth at that time and then written into the database.

Another essential function was to click again on a seat to cancel the selection. This was implemented by using an array to store the seat's UUID. All reactions of a seat being tapped is shown by the pseudocode below:

when a seat being tapped,
if seat exists{
    if have not selected enough seats && seat is not taken{
        put seat's id in arr[]
        seat.taken = true
    }else if (seat is taken && seat's id is in arr[]){
        remove seat's id from arr[]
        seat.taken = false
    }else{return}
else{return}

Also, if the user chose n tickets in the previous page, but he/she did not select enough seats, when clicking confirm, the system would alert. Only when selecting exactly the same number of tickets, the user was allowed to confirm his/her order. (Figure) This function was achieved by comparing the seatNum passed from the previous page with

the number of UUID in the selected seat array.

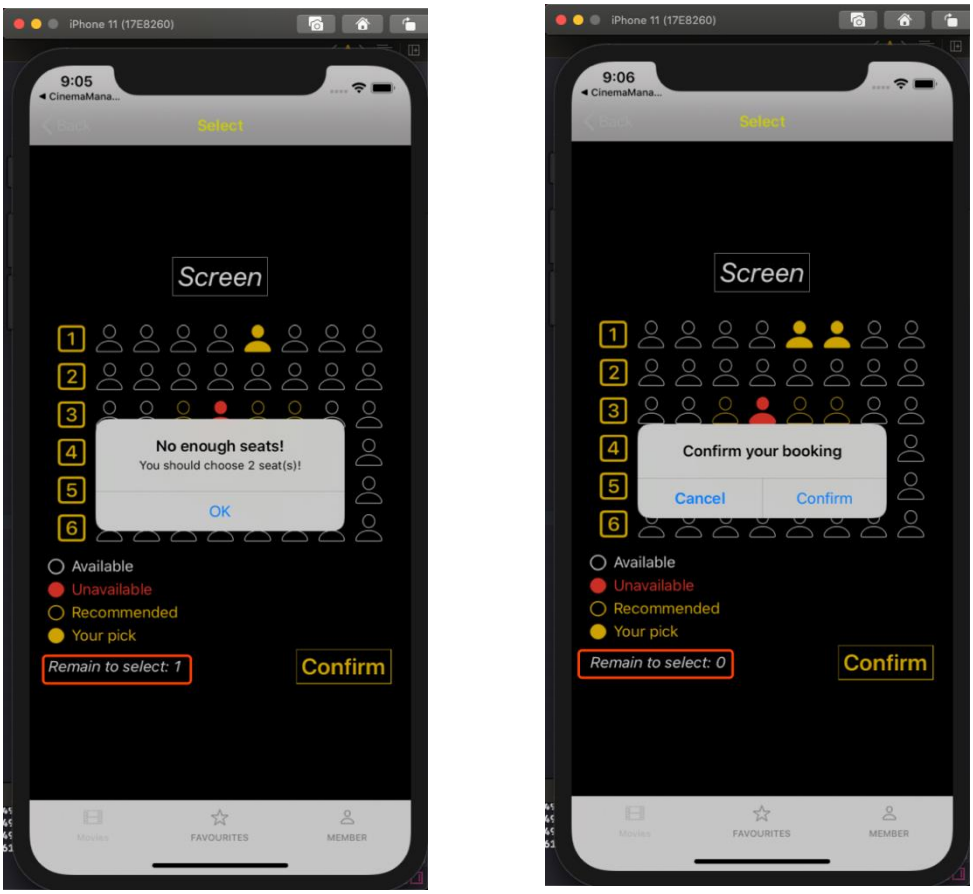

Figure 38: Select enough seats before confirm

When the user click on "Confirm" again, the book() function would be executed and generate a ticket object, containing the movie, date, time, and seat information.

```
func book(){
    var bookingSeats:[Seat] = []
    for id in mySeat{
        let singleResult = self.realm?.objects(Seat.self).filter("id = '\(id)'").first
        bookingSeats.append(singleResult!)
    }
    let ticket = Tickets(Screenid: self.screen.screenId, moviename: self.screen.moviename, date: self.screen.date,
                         time: self.screen.time, seats: bookingSeats, username: self.regularuser.username)
    let user = realm?.objects(User.self).filter("username = '\(self.regularuser.username)'").first
    try! self.realm!.write{
        user?.orders.append(ticket)
        for seat in bookingSeats{
            seat.taken = true
        }
    }
}
```

1

2

Figure 39: Book seats

In block 1, the seat objects in the database were queried by the UUID of the copied FakeSeats, which were the same with those authentic ones. In block 2, a ticket object was appended to the list of tickets named orders and the real seats' taken Boolean were changed to true.

3.5.9 Seat preferences

The seat preferences could be set in the user's page after logging in. For the horizontal preference, center and side could be chose and for the vertical preference, there were front, middle and back. After the user select his/her preferred seat position and clicked "Done" on the trailing top corner, the database would be updated and recommended suitable seats the next time he/she entered the select seat view.

Since the preferences of the seats were set upon their initialization, the cinema could change it according to their seat plan. For example, now, the system was designed to be 6 row and 8 seats in each row. The front, middle, back corresponds to 1,2 rows, 3,4 rows, and 5,6 rows respectively. If there were more rows, the preferences plan could be changed conveniently.

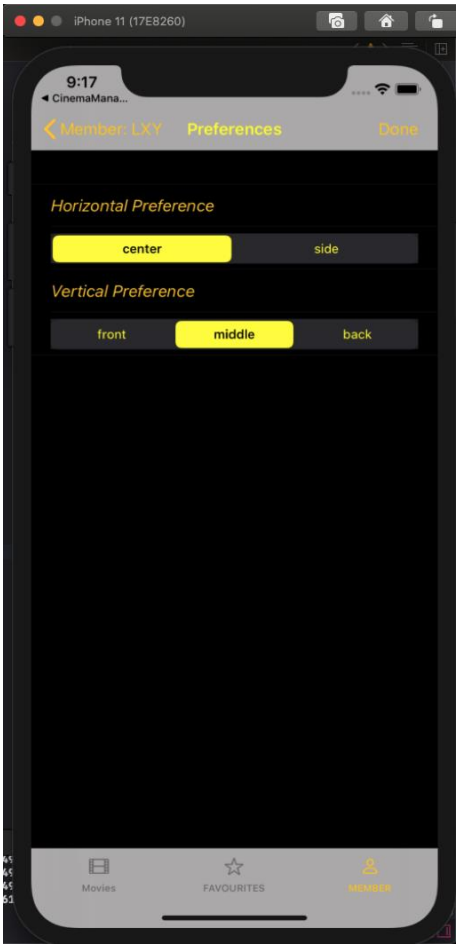

Figure 40: Preferences setting page

3.5.10 Tickets

In the user page, the user could also view his/her bought tickets and could cancel them if needed.

The UI is shown in Figure 41, displaying the situations:

1) No tickets
2) Some tickets

3) Cancelling tickets

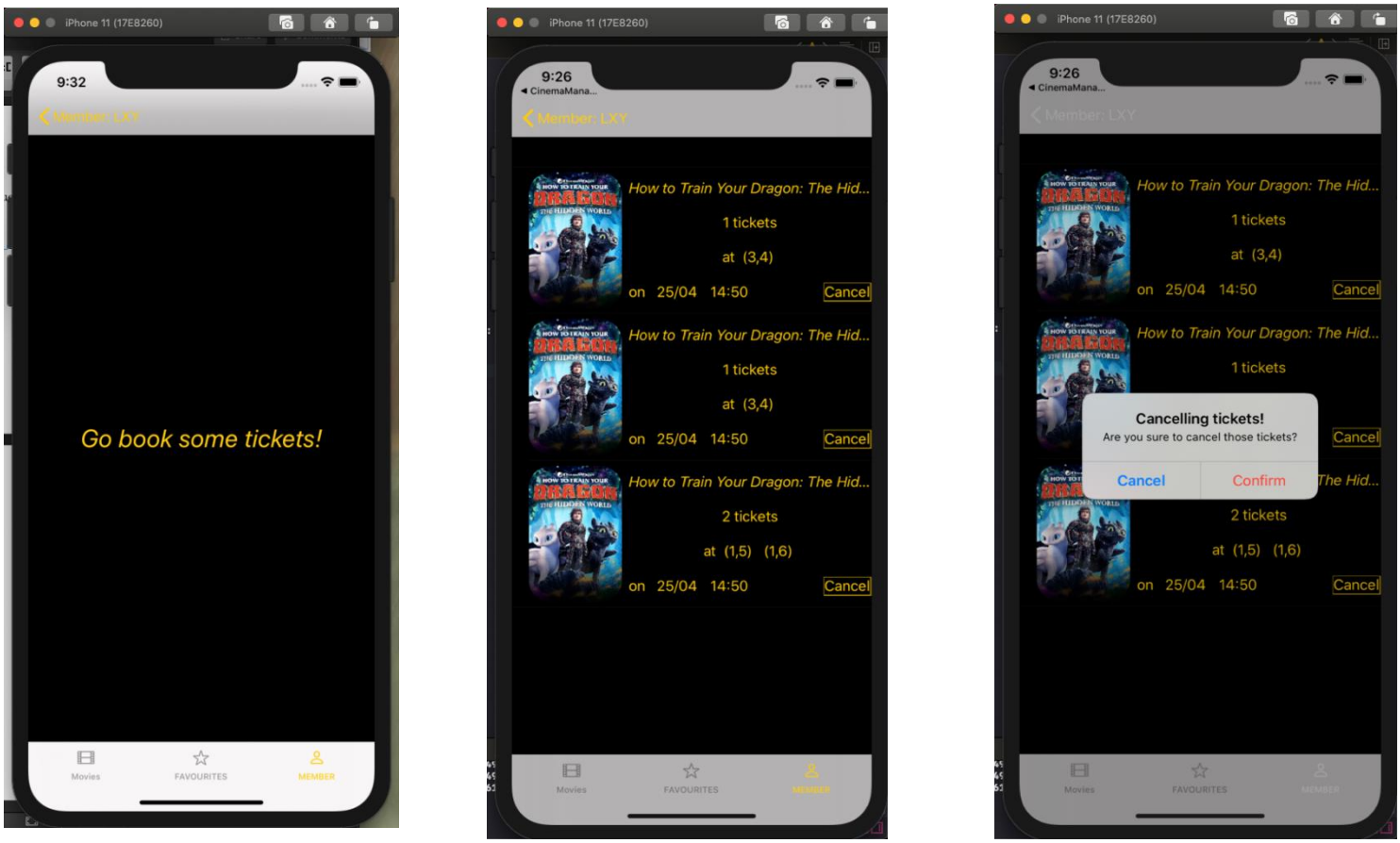

Figure 41: No tickets, Three orders and Cancelling an order

The problem encountered while coding for this page was similar to the one in SelectSeatView. The user could choose cancel the tickets and regret at the alert window. So myFakeTicket were also used to avoid directly manipulating the realm object.

4 Testing & Evaluation

4.1 Software Test

4.1.1 RealmExp App

To test the functions of this app, the database needed to be initialized and it would be hard if all data was added manually. So, another app called RealmExp was developed to initialize the database. Fill in some movies, stars, screens and so on. This app was

relatively simple and focusing on the code part.

```
func generateSeats(screenId: Int,moviename:String,time:String)->[Seat]{
    var seatArray:[Seat] = []
    if screenId == 1{
        for i in 1...6{
            for j in 1...8{
                seatArray+=[Seat(moviename: moviename, date: "25/04", time: time, row: i, column: j,
                    exist: true)]
            }
        }
    }
    if screenId == 2{
        for i in 1...6{
            for j in 1...8{
                let seat = Seat(moviename: moviename, date: "25/04", time: time, row: i, column: j,
                    exist: true)
                if (i<=2&&j<=2)||(i==6&&j==8){
                    seat.exist = false
                }
                seatArray+=[seat]
            }
        }
    }
    return seatArray
}
```

```
func writeSampleScreenforMovies(realm:Realm){
    let movienames = ["How to Train Your Dragon: The Hidden World","The Wolf of Wall
        Street","Inception"]
    let time = ["14:50","17:30","20:20"]
    for i in 1...2{
        for j in 1...3{
            let screen = Screen(screenId: i, moviename: movienames[j-1], date: "25/04", time:
                time[j-1], seats: generateSeats(screenId: i, moviename: movienames[j-1], time:
                time[j-1]))
            try! realm.write{
                realm.add(screen,update: .all)
            }
        }
    }
}
```

Figure 42: Generate seats and generate Screens

Throughout the whole test process, including unit bug test and the final tests with human volunteers, there were several times that the database was ruined by the Movie app, which either configured the wrong object or written a wired value or could not delete a ticket and so on. This RealmExp app helped reduced time wasted and if any of the mentioned accident occurred, the developer could just delete the whole database and re-initialize it with several clicks.

4.1.2 Test Along Development and By Volunteers

As an app, the bugs normally would be transition between the pages going wrong or that the user clicked but the app acted like not. These bugs were tested and solved along the development. However, the human volunteers did provide some interaction bugs such as that the user could confirm a set of 3 seat selections even if he/she actually chose to buy 5 tickets. Those bugs were taken care of as well, benefiting from the prototype development model.

4.2 Evaluation

During the tests along with the development, the developer was satisfied with the app performance. The app was able to transit between different pages smoothly and the connection with the cloud server was stable. The transmission speed of data, the loading speed of views were all satisfying. However, the participants of the beta version test provided some valuable advices, which the developer was not able to implement due to the time limitation. But their words were summarized and recorded for future improvement of the app, or similar projects.

Most participants liked the UI design of the movie detail page and the all movies page, feeling that the two colors, black and yellow are suitable for a cinema app and make people think of movies. However, staring at the app for a relatively long time might be bad for eyes because it would force the users to raise the lightness, and when they switch to another app, the screen was usually too bright.

Over 50% of the testers wanted the rating and review function, to let them sharing their thoughts towards the movies. The idea was reasonable and widely used in the existing app, but the developer did not implement it.

Overall, the app was a reasonable one with all basic functions of a cinema app, although there were still improvements could be made.

5 Conclusion

Look back to the whole project, the aims and objectives were mostly well implemented in the app.

The project aimed to build an app that allowed the user to view movie information, book seats on their iOS devices and recommended seats according to user's preferences. The app successfully realized the first two aims but for the rest two aims, it did not perform well. The manager of the cinema surely could manage the seat plans and change information through the current system, but he/she would need to use the backend of the database to do so. It was a flame that the app for manager was not yet developed to form a complete system on mobiles. Also, the function that seats were recommended by algorithm instead of user's settings was not implemented, although all objectives were successfully achieved. The app contained abundant user interfaces and could interact with the users well. It could run on different iOS devices and the information in user account could be synced. Moreover, the data downloading speed was satisfying and the user merely did not have to wait while switching between different views.

6 BCS Criteria & Self-Reflection

- The developer learnt SwiftUI from scratch and read document about all kinds of cloud server. Finally choosing Realm and successfully built the app showed that the developer had used the most precious thing learnt from the university to address the problems—the ability to learn on his/her own, even without the help of others. The coding ability also rose through the project.
- The innovation in this project was the way the developer solving the problem that realm object could not be amended outside the framework of realm. Using "fake" object and converting them to real ones when needed was a good move.
- The app met the demand of market. As a cinema app, having the most functions

that apps on the market have, this app had its own commercial value.

- The time management of this project was not satisfying. The developer spent too much time on other subjects and did not realize the difficulty lay on this project. Nearly two months before the deadline, he started to work and found he had a lot to learn. If he had had some more time, the app could be more complete and more satisfying. For example, the user rating and reviewing functions were not added dur to the time issue. However, the hard work was worth praising, and the developer's interests in the project were the shining side.
- Overall, the goal of the project was mostly achieved and the developer had learnt through the progress, not only on the coding skill aspect, but also on the project management aspect and learning skill aspect.

7 References

ODEON on the App Store (2020). Available at: https://apps.apple.com/gb/app/odeon/id416549949 (Accessed: 14 May 2020).

Realm (2020). Available at: https://github.com/realm (Accessed: 14 May 2020).

SF Symbols (2020). Available at: https://developer.apple.com/design/human-interface-guidelines/sf-symbols/overview/ (Accessed: 14 May 2020).

SwiftUI (2020). Available at: https://developer.apple.com/xcode/swiftui/ (Accessed: 14 May 2020).