IOS App zur Anzeige von TV-Programmempfehlungen

unter Verwendung einer HTTP-Webdienst-API

Carolina da Rocha Nobre
EI18WI-b

Abschlussbeleg für das Lehrfach
Mobile Development Application (IOS)

Fakultät Medien
Prof. Rico Thomanek, M.Sc.
Hochschule Mittweida
WS2020

Contents

1 Übersicht

1.1 Aufgabenbeschreibung

Ziel dieses Projekt ist die Implementation ein IOS Applikation zur Anzeige von TV-Programmempfehlungen unter Verwendung einer HTTP-Webdienst-API. Dafür wird ein Rechner mit dem MacOS Betriebssystem benötigt und das Software Xcode installiert. Für dieses Projekt wurde Xcode Version 12.4 (12D4e) verwendet.

Die Implementation dieser App wird mittels zwei verschiedene Frameworks implementiert werden. Das erste Teil dieser Beleg wird sich mit der Implementierung mittels Storyboard befassen. Storyboard ist einem Drag-and-drop Interface Builder innerhalb Xcode, das zusammen mit dem Framework UIKit implementiert wird.

Das zweite Teil dieser Beleg beschreibt, wie man das gleiche Interface mit dem SwiftUI Framework implementieren kann. Der SwuiftUI Framework ist ein neues Framework, was uns ermöglicht den ganzen User Interface mit weniger Code zu erstellen. Im Gegensatz zu UIKit, ist SwiftUI komplett Software-basiert d.h. man braucht keine Storyboards. Die SwiftUI Syntax ist sehr einfach zu verstehen und ein SwiftUI-Projekt kann mit der automatischen Vorschau schnell betrachtet werden.

Um die benötigten Programmempfehlung zu erhalten wurde uns zwei URL gegeben. Einmal im XML-Format und einmal in JSON-Format. Hier habe ich das URL in JSON-Format verwendet.

2 Implementation mit Storyboard

XCode Version	12.4 (12D4e)
Frameworks	UIKit Foundation

2.1 Neues Projekt erstellen

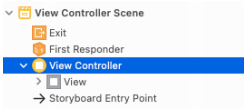
In der Welcome to Xcode Bildschirm geht man auf: Create a new Xcode project Project Template: App Options: Name: BelegStoryboard Interface: Storyboard Language: Swift

nicht genannte Einstellungen müssen nicht geändert werden.

In das generierte Projekt kriegt man Automatisch eine ViewController.swift File und in das main.storyboard File hat man eine View Controller vorgegeben. Für dieses Projekt werden diese nicht benötigt deshalb kann man Das View-Controller.swift File löschen.

control + klick auf die ViewController.swift File > Delete > Move to Trash.

In das main.storyboard File, View Controller löschen. Am besten kann man in der outline view auf das View Controller klicken und die delete Taste drucken (Bild 1)



1: outline view

Anstatt das View Controller wird eine Table View Controller verwendet. File > New > File > Cocoa Touch Class

Class: ProgramTableViewController Subclass of: UITableViewController Language: Swift

Die Cocoa Touch Class File ist eine Swift File, kommt aber schon vor-programmiert mit dem benötigte Funktionen für das ausgewählte Subclass.

Jetzt zurück zu dem main.storyboard müssen wir aus dem Library einem Table View Controller holen und in dessen Identity Inspector unter Class unsere neue ProgramTableViewController auswählen.

2.2 Zellen Design

Zum Schluss soll das Table View Controller wie auf Bild 2 aussiehen.

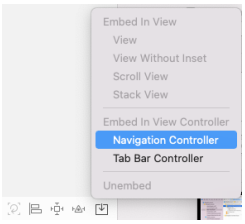
Um die Zelle zu entwerfen, habe ich zuerst ein Bild in das Programm importiert zu bessere Übersicht. Das Bild soll am besten die große 1280x720 Pixels haben, weil die Bilder in der JSON File auch diese große haben. In der Project Navigator öffnet man Assets.xcassets und kann einfach aus dem Finder ein neues Bild in das Projekt ziehen.

In der main.storyboard wird der Table View Controller jetzt in einem Navigation Controller eingebettet (Bild 3. Diese schritt ist nicht notwendig, da die Applikation nur einem View hat, das Titel sieht aber so besser aus.

In die neue Navigation Item > Attributes Inspector> Title: ”Kodinerds freeTVAnnouncement Service” ändern.

Jetzt in die Zelle eine Vertical Stack View ziehen. Alle Constraints = null setzen. In die Vertical Stack View eine Image View und 4 Labels ziehen.

2: Zell



3: Navigation Controller einbetten

Konfiguration für das Image View: in die Attributes Inspector > Image: image (importiertes Bild der Assets.xcassets) in verschiedene Geräte sieht das Bild nicht optimal aus, deshalb habe ich auch die Constraint aspect ratio hinzugefügt. control + ziehen von Bild zu sich selbst, aspect Ratio auswählen, dann in die Size Inspector, in die neue erstellte Constraint Edit auswählen und multiplier zu 414:233 setzen.

Für die Labels auch am besten irgendeinem Default Text eingeben zur besseren Übersicht. In die jeweiligen Attribute Inspector müssen für alle Labels Lines: 0 gesetzt werden, damit der Text über mehrere Zeilen gehen kann und der Text View sich automatisch anpasst. Dazu müssen noch die Fonts von Label 1 zu Title 2 und Label2 zu Title 3 geändert werden.

Somit sind die Zellen fertig.

2.3 Program Item Modell

Um dem Inhalt von der JSON Data in unsere Zell zu anzeigen, müssen wir einem Modell Item erzeugen. Dafür habe ich eine neue Swift Datei ProgramItem.swift erzeugt.

In dieser Datei wird erstmal ein struct ProgramItem erzeugt. Diese struct muss das Protokoll Codable adoptieren. Das Codable Protokoll macht Datentypen kodierbar und dekodierbar für die Kompatibilität mit externen Darstellungen wie JSON.

In das Protokoll muss für jedes Label und das Image View eine variable von der optional Type String? deklariert werden. Der Type muss optional sein, weil es könnte sein, dass eine bestimmte Key kein Value paar liefert. Deshalb muss es erlaubt sein, dass die variable auch nil sein kann.

Jetzt braucht die struct eine Initialisierer. Den habe ich $\mathrm{init}(\mathrm{programDictionary:})$ genannt. Dabei habe ich auch eine Instanz $\mathrm{program}$ der Typ [String:Any] erzeugt. Die JSON Data hat in alle Keys Strings, aber für den Value gibt es auch z.b. unter Broadcasts einen anderen Key-Value paar mit einem Array als Value. Deshalb muss man als Eingabe einem Dictionary mit Strings als Keys und Any als Value. $[\mathrm{String:Any}]$

Zu bessere Übersicht der JSON Data habe ich den JSON Text aus dem `https://www.staff.hs-mittweida.de/~rthomane/grdl_iphone/knftv/broadcasts.php` URL in dem Webtool `https://jsonformatter.org/` kopiert. Diese Website formatiert das JSON Text in einem besseren lesbare Textdatei. So sieht den Text nach dem Formatieren aus:


```
{
  "events": {
    "knftv": [
      {
        "Nickname": "Boogie2005",
        "uTime": "1622744100",
        "ChannelName": "ServusTV HD Deutschland",
        "Icon": "https://www.staff.hs-mittweida.de/~jesch/knftv/cache/1622101618",
        "IconType": "0",
        "Date": "2021-06-03 20:15",
        "StartTime": "20:15",
        "EndTime": "21:40",
        "RunTime": "85",
        "Title": "Der Gott des Gemetzels",
        "EpgEventTitle": "Der Gott des Gemetzels",
        "Genre": "Andere / Unbekannt",
        "Plot": "Schauspieler: Jodie Foster ...",
        "Rating": [],
        "Broadcasts": {
          "Broadcast": [
            "2021-06-03 18:15",
            "2021-06-03 21:30"
          ]
        },
        "Recommendations": {
          "User": "Boogie2005"
        }
      },

      ...
```

2.3.1 Type Casting

In der Initialisierer Körper werden die Values aus dem JSON Dateien geholt und in der struct variablen zurückgegeben. Weil die Values von der Any Typ sind, bevor wir die Dateien benutzen können, müssen wir prüfen was für ein Typ die Values haben. Das wird mit dem as? Operator realisiert. Im Prinzip funktioniert das Type-Casting immer ähnlich, daher werde ich hier nur einmal mit den code Zeilen 36-40 erklären.

```
guard let channelName = program["ChannelName"] as? String,
      let title = program["Title"] as? String,
      let runTime = program["RunTime"] as? String else {return}

self.title = "\(String(describing: date)) auf \(channelName): \(String(describing: title))
    (\(String(describing: runTime))min)"
```

Persönlich finde ich die guard let Anweisung besser als ineinander verkettete if Anweisungen.

Wenn das Key ”ChannelName” eine String zurückliefert, diese String in die konstant channelName speichern und nächste Zeile lesen.

Sonst: Return

Wenn das Key ”Title” eine String zurückliefert, diese String in die konstant title speichern und nächste Zeile lesen.

Sonst: Return

Wenn das Key ”RunTime” eine String zurückliefert, diese String in die konstant runTime speichern und nächste Zeile lesen.

Dann mit dem erhaltenen Informationen als String in title zurückgeben.

Zeile 52-60: In der JSON File kommt nach ”Broadcasts” keine String, sondern andere Key Value paar, deshalb habe ich hier einmal das Key ”broadcast” mit einem as? [String:Any] zugegriffen. In das Broadcast Key können zwei Values Typen kommen. Eine String Array [String] oder bloß einem String.

```
"Broadcasts": {
  "Broadcast": [
    "2021-06-03 18:15",
    "2021-06-03 21:30"
  ]
},
```

[String]

```
"Broadcasts": {
  "Broadcast": "2021-06-03 18:15"
},
```

String

Man kann die Strings Array mittels joined(separator; ”, ”) in einem einzigen String konvertieren. Falls bloß einem String gefunden wird, einfach diese String zurückliefern.

Zeile 62-63: Merke, dass in der iconUrl wird nicht das Bild direkt zurückgeliefert, sondern nur die URL des Bildes.

2.3.2 Unix Time

Zeile 22-34: Das Datum wird in der Format Unix Time angegeben. Die Unix Time zählt die Sekunden seit 01.01.1970 um 00:00:00 auf 0° Längengrad auf. Deshalb müssen wir aus dem Unix Time in einem besseren menschlichen lesbare Format konvertieren. Am besten WW. TT.MM.JJJJ SS:MM in Deutschland Zeitzone (+1, aktuell Sommerzeit, also, +2). Dafür müssen wir den Framework Foundation importieren, welches die Datentype Date enthaltet und auch Funktionen für Daten und Zeiten Umrechnungen.

In die Framework Foundation hat man einem struct Date mit dem Initialisierer $\mathrm{init(timeIntervalSince1970:\ TimeInterval)}$, welches eine wert relative zu 00:00:00 am 01.01.1970 UTC (Coordinated Universal Time) erzeugt [3]. Da diese struct ein Double nimmt und wir einem String aus dem JSON Datei erhalten, müssen wir zunächst diese String Datei in einem Double konvertieren. Mit diesem Double rufen wir die struct mit dem obengenannten Initialisierer.

Der Foundation Framework hat auch einem $\mathrm{DatenFormatter}$ Klasse [4]. Mit dieser Klasse kann man den Standort sowie das String-Format definieren. Das Format wird dann in der $\mathrm{dateFormatter}$ variable gespeichert. Zu Schluss rufen wir die Funktion $\mathrm{date.string(from:\ )}$ mit dem Parameter $\mathrm{currentDate}$ und diese String wird in Date gespeichert. Das variable date muss außerhalb der $\mathrm{if\ let}$ Schleife deklariert werden, denn wir es später an der Title benutzen werden.

Zeile 44: hier habe ich zwei Leerzeichen ” ” mit einem ”\n” ersetzt. In Markdown Auszeichnungssprache werden zwei Leerzeichen zu Zeilenumbruch verwendet. Der JSON File ist wahrscheinlich an diesem Format gerichtet. Zeilenumbruch in einem Swift String wird mit \n gemacht, deshalb dem Ersatz.

2.4 Table View Cell

File > New > File > Cocoa Touch Class Class: ProgramTableViewCell Subclass of: UITableViewCell Language: Swift

Erstens muss die klasse in die Identity inspector der Zellen zu den neuen erstellte Klasse und auch die Restoration ID zu ”programCell” geändert werden. Die Restoration ID wird für die Zelle Dequeueing verwendet.

Eine Table View kann tausende Zellen haben. Die Zellen müssen aber in eine endliche Handy speicher existieren. Wenn die Table View versuchen wurde, für jedes Objekt eine andere Zelle zu laden, würde eine große Liste schnell dazu fuhren, dass dem gerät der speicher ausgeht, was zu einem Absturz führen würde. Um dieses Problem zu beheben, laden Tabellenansichten nur die sichtbaren Zellen und ein paar mehr um sicherzustellen, dass das Scrollen flüssig bleibt. Normalerweise haben die Zellen alle das gleiche Layout. Die neue Zellen können dadurch die alte Zellen wiederver-

wenden. Diese verfahren wird Cell Dequeueing genannt[1].

Dazu muss man mittels control + ziehen die @IBOutlets für alle Labels und der Image View erstellt werden.

2.5 Table View Controller

Hier werde ich die Implementation in die File ProgramTableViewController.swift erklären.

2.6 JSON File herunterladen

Seit IOS 9 müssen alle Verbindungen über HTTPS erfolgen. HTTPS ist http mit Verschlüsselung. Daher ist HTTPS sicherer. Falls man trotzdem http benutzen will, muss man in der Info.plist File das explizit angeben. In die Projektnavigator > BelegStoryboard > Info > + auf Bundle OS Type code > App Transport Security Settings > Allow Arbitrary Loads Value: YES. [17]

Die DownloadTask wird in dem viewDidLoad Funktion implementiert. Das viewDidLoad wird gerufen, sobald das Haupt View eines ViewControllers aufgeladen ist.

```
10 class ProgramTableViewController: UITableViewController {
11
12     var programArray: [Any] = []
13
14     override func viewDidLoad() {
15         super.viewDidLoad()
16
17         let downloadTask = URLSession.shared.downloadTask(with: URL.init(string:
               "https://www.staff.hs-mittweida.de/~rthomane/grdl_iphone/knftv/broadcasts.php")!) { (location,
               response, error) in
18             if let data = try? Data.init(contentsOf: location!){
19                 if let programDictionary = try? JSONSerialization.jsonObject(with: data) as? [String:Any] {
20
21                     if let broadcast = programDictionary["events"] as? [String:Any]{
22                         self.programArray = (broadcast["knftv"] as! [Any])
23                     }
24
25
26                     DispatchQueue.main.async {
27                     self.tableView.reloadData()
28                     }
29                 }
30             }
31         }
32         downloadTask.resume()
33     }
```

Für das Download wird die Klasse URLSession verwendet [14]. Die klasse URLSession hat einem singleton shared object, welches eine URLSession-Instanz liefert. Die shared Session ist für Basisanforderungen geeignet. Es hat viele Einschränkungen, da es sich nicht konfigurieren lasst, reicht aber für unsere Anwendung aus. [13]. Innerhalb der shared Session wird dem downloadTask Methode gerufen. Das downloadTask(with:completionHandler:) ruft Daten in Form einer Datei ab [5]. Diese Funktion erwartet eine URL und ein completionHandler Closure.

Für die erste Parameter geben wir die String "`https://www.staff.hs-mittweida.de/~rthomane/grdl_iphone/knftv/broadcasts.php`". Wir müssen aber

erst die String mit dem URL.init(String:) in einer URL umwandeln. Diese Initialisierer liefert einem optional zurück, daher muss die URL mit dem Operator ! ausgepackt werden [9].

In dem completionHandler kriegt man 3 variable: error wird gleich nil, wenn die anfrage erfolgreich wurde; location ist die temporär Speicherort der heruntergeladenen Datei. Die Datei musst innerhalb des Closures irgendwo anderes kopiert werden, danach wird die Datei gelöscht; Das response Objekt stellt Antwort-Metadaten bereit, z. B. HTTP-Header und Statuscode.

Die erste Zeile innerhalb der Closure prüft, ob eine Datei in location vorhanden ist. In dem fall, wird diese Datei in der konstant data kopiert und der code innerhalb der if let Closure wird gelesen. Da diese Funktion Fehler auslöst, muss sie mit try? aufgerufen werden(throw funktion).

Innerhalb der if let Closure initialisieren wir die Data mit init(contentsOf:options:) [8] und speichern sie in die variable data. Die contentsof parameter fragt, wo meine Dateien sich befinden, hier sind sie in der variable location. Options ist hier nicht relevant, deshalb lasse ich es einfach weg. Wenn das funktioniert, muss man die JSON Datei in Foundation Objekte umwandeln. Dafür wird die klasse JSONSerialization verwendet [11]. Foundation Objekte sind in der Framework Foundation enthalten, die in der Datei Programitem.swift schon importiert wurde.

Um eine neues JSON Objekt zu erstellen, braucht man die Funktion JSONObjekt innerhalb der klasse JSONSerialization [10]. Diese Funktion nimmt zwei Parameter. Die erste Parameter eine Data, wir nehmen die Data, was ich in data gespeichert habe. Der zweite Parameter ist in dieser Anwendung irrelevant und daher habe ich es weg gelassen. Zum Schluss wird den Type Casting Operator as? verwendet um zu prüfen, ob die JSON Data der Art [String:Any] ist. Falls eine JSON Data in diesem Format gefunden wurde, wird der code innerhalb der if let Anweisung gelesen.

Hier funktioniert das Auspacken ähnlich wie in der ProgramItems struct.

Falls der Key "events" der Typ [String:Any] das in der variable broadcast speichern.

falls der Key (innerhalb broadcast) ein [Any] Array ist, das in der programArray speichern. Hier wollen wir keine Optional zurückliefern, deshalb wird der Operator as! anstatt as? für den Type Casting verwendet. Der Operator as! prüft die variable und erzwingt das Ergebnis als eine einzige zusammengesetzte Aktion. Man sollte die ! Operator nur verwenden, wann man ganz sicher von der Ergebnisse ist, da ein Auspacken Fehler zu program Abbruch führen wurde.

programArray muss als global Variable deklariert werden, denn wir benutzen es in der ProgramItems struct.

Zum Schluss wird die Tabelle aktualisiert mit:

Nachdem der Task erstellt wurde, muss man sie durch Aufruf ihrer resume() Methode starten.

```
DispatchQueue.main.async {
self.tableView.reloadData()
}
```

2.7 Zelle mit daten befüllen

Jetzt werden wir die vorgegebenen Funktionen der Cocoa Touch Class Table View ausfüllen.

Als erstens müssen wir die Funktion numberOfSections ändern. Da wir nur eine Sektion in unsere Tabelle haben, müssen wir einfach 1 zurückgeben.

```
override func numberOfSections(in tableView: UITableView) -> Int {
    return 1
}
```

Die nächste Funktion numberOfRowsInSection gibt zurück wie viele Zeilen wir möchten. Die Anzahl an Zeilen soll das Gleiche sein wie die Objekte Anzahl in unsere programArray.

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return programArray.count
}
```

Dann muss man die cellForRowAt Funktion implementieren. Diese Funktion wird so viele mal aufgerufen, wie unsere Anzahl an Zeilen. Jedes Mal konfiguriert es eine neue Zelle mit einem neuen indexPath.

Als erstens muss das cell Dequeueing programmiert werden.

Das IndexPath ist einem struct [7], und innerhalb dieser struct gibt es eine variable row der Typ Int. Speichere diese row int in index um die Zelle in der richtige Zeile zu konfigurieren.

Dann nehmen wir aus dem programArray nur die Key-Value paar in die stelle index. Dieses Key-Value paar wird an der Typ ProgramItem konvertiert und in der variable program gespeichert.

Zuletzt muss man die entsprechenden Strings aus der program variable in die Zellen Label zuweisen. Das iconURL String wird in einer URL konvertiert, danach wird geprüft ob irgendeine Data in diese URL liegt. Diese Data wird an die Image View der entsprechenden Zell zugewiesen.

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "programCell", for: indexPath) as!
        ProgramTableViewCell
    let index = indexPath.row
    let program = ProgramItem(programDictionary: programArray[index] as! [String:Any])
    // Configure the cell...
    cell.title?.text = program.title!
    cell.details?.text = program.details!
    cell.programDescription?.text = program.description!
    cell.subtitle?.text = program.subtitle!

    let url = URL(string: program.iconURL!)
    let data = try! Data(contentsOf: url!)
    cell.programImage?.image = UIImage(data: data)


    return cell
}
```

Somit ist das Storyboard Implementierung fertig.

3 Implementation mit SwiftUI

XCode Version	12.4 (12D4e)
Frameworks	UIKit SwiftUI

3.1 Create new Project

Mit einem XCode Projekt geöffnet, file > new > project > App Name: Interface: SwiftUI Language: Swift

3.2 Program Model

Das Program Model ist das gleiche, wie in der Storyboard-Implementierung. Daher wird hier eine neue Swift File erzeugt: File > new > File > Swift File > Save As: ProgramItem.swift

Das Model muss, damit es mit den Zellen funktionieren, das Protokoll Identifiable übernehmen.

Um das Protokoll Identifiable konform zu sein, braucht man einem variable id und alle variable müssen initialisiert werden. Das Variable id wird mit dem Funktion UUID() initialisiert. Alle andere Variable werden mit nil initialisiert.

```
struct ProgramItem: Codable, Identifiable {

    var id = UUID()
    var title: String? = nil
    var subtitle: String? = nil
    var description: String? = nil
    var details: String? = nil
    var iconURL: String? = nil
```

Außerdem habe ich eine Funktion $\mathrm{myImage}$ hinzugefügt. Diese Funktion nimmt der String in $\mathrm{iconURL}$. Diese String wird in einer URL konvertiert. Falls irgendeinem Data in das gegebenen URL gefunden wird, wird das gefundenen Data zurückgegeben. In das Storyboard habe ich diese code Zeilen direkt in der $\mathrm{cellForRowAt}$ implementiert.

```
func myImage (iconUrlString: String) -> Data {
    let url = URL(string: iconUrlString)
    let data = try! Data(contentsOf: url!)
    return data
}
```

3.3 downloadData

Auch hierfür habe ich einen neuen File Erzeugt: File > New > File > Swift File > Save As: downloadData.swift.

Die DownloadData Methode muss das Protokoll ObservableObject adoptieren. Dann brauchen wir 3 Variable: programArray:[Any]? und url: String sind schon aus dem Storyboard Programm bekannt. Neu dazu ist die Variable programItems:[ProgramItem]. Diese Variable hat den Property Wrapper @Published. Das Wrapper gehört zu dem ObservableObject Protokol. Diese Variable schickt Meldungen immer, wenn programItems geändert wird [15].

Danach wird noch ein Initialisierer hinzugefügt, damit man die Classe aufrufen kann. Das Initialisierer nimmt ein String als Parameter, konvertiert das String in einer URL und gibt dieses zurück in dem url Variable.

```
class downloadProgram: ObservableObject {

    var programArray:[Any]? = nil
    var url: String = ""
    @Published var programItems = Array<ProgramItem>()

    init(programURL:String?) {
        if let programURL = programURL {
            url = programURL
        }
    }
```

Das downloadTask erfolgt gleich wie in der Storyboard-Implementierung. Danach werden die Zellen aktualisiert.

Bei dem Storyboard haben wir den cellForRow Schleife verwendet um die Zellen zu befüllen. Hier wird erstmal geprüft, ob da programArray nicht nil ist. Falls das stimmt, werden wir bei jeder Wiederholung einem neuen item der Typ ProgramItem initialisieren. Danach wird das item in dem Array programItems angehängt.

Weil das programItems den Wrapper @Published hat, kann der Task nicht im Hintergrund geändert werden. Daher müssen wir die Zeile in einem DispatchQueue.main.async einbetten [2].

```
if self.programArray != nil {
    for program in self.programArray! {
        let item = ProgramItem.init(programDictionary: program as! [String:Any])
        DispatchQueue.main.async {
            self.programItems.append(item)
        }
    }
}
```

3.4 Zellen

Jetzt werden die Zellen entworfen.

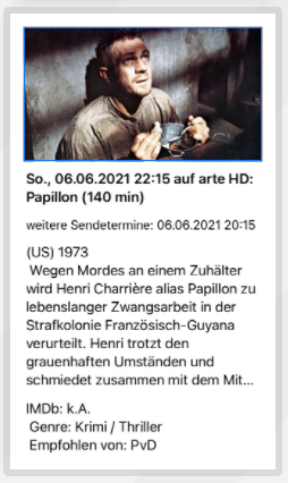
File > new > File > Swift UI View > Name: BroadcastRow.

In meinem Xcode Projekt habe ich die File BroadcastRowTemplate.swift generiert, um diesen Schritt besser darzustellen. Sie ist aber für das Projekt nicht relevant.

Die SwiftUI View kommt schon mit zwei Structs. Das BroadcastRow und das BroadcastRow_Previews. Damit die Vorschau schöner aussieht, kann man in das BroadcastRow_Previews zwei Attribute implementieren. Das .previewLayout(.sizeThatFits), damit den Preview grenzen sich zu dem Inhalt anpasst und das .padding, damit die grenzen mit ein bisschen Abstand von dem Inhalt platziert werden.

```
struct BroadcastRowTemplate_Previews: PreviewProvider {
    static var previews: some View {
        BroadcastRowTemplate()
            .previewLayout(.sizeThatFits)
            .padding()
    }
}
```

Jetzt wird die Implementation innerhalb BroadcastRow erklärt. Wir brauchen eine ImageView und vier Labels. Zum vorschau habe ich einfach irgendeinem Text in die Labels hinzugefügt. Für Image hatte ich in der assets.Xcassets ein Bild importiert. Das Bild werde ich nur zum Entwerfen benutzen.



4: Zell

Die Zellen sollen am Ende wie auf dem Bild 4 aussehen.

Für die Image braucht man einem .rezizable Modifier, damit man das Bild Größe ändern darf. Dann muss man den Content Mode mit den Modifier .aspectRatio(contentMode: .fit) zu Fit ändern damit der Größe des Imageträgers sich an der Schirm anpasst. Und zuletzt braucht man noch einem .scaleToFit Modifier, damit das Image innerhalb des Imageträgers sich selbst skaliert um den ganzen Container zu befüllen.

Die Labels werden die Modifier .frame(maxWidth: .infinity, alignment: .leading) und .padding(3) haben. .infinity damit die Labels das ganze Schirmbreite belegen, .leading damit die Texte links anfangen, und .padding() damit sie mit ein bisschen Abstand von der Schirmgrenzen und auch von dem andere Labels platziert werden. Für den title Label habe ich der Font .headline verwendet. Für den subtitle Label habe ich der Font .subheadline verwen-

det. Für die zwei andere Labels habe ich der Font .body genommen.

```
struct BroadcastRowTemplate: View {
    var body: some View {
        VStack {
            Image("papillon")
                .resizable()
                .aspectRatio(contentMode: .fit)
                .scaledToFit()
            Text("So., 06.06.2021 22:15 auf arte HD: Papillon (140 min)")
                .font(.headline)
                .frame(maxWidth: .infinity, alignment: .leading)
                .padding(3)
            Text("weitere Sendetermine: 06.06.2021 20:15")
                .font(.subheadline)
                .frame(maxWidth: .infinity, alignment: .leading)
                .padding(3)
            Text("(US) 1973 \n Wegen Mordes an einem Zuhälter wird Henri Charrière alias Papillon zu
                lebenslanger Zwangsarbeit in der Strafkolonie Französisch-Guyana verurteilt. Henri
                trotzt den grauenhaften Umständen und schmiedet zusammen mit dem Mithäftling Dega
                einen gewagten Fluchtplan. Doch der erste Fluchversuch schlägt fehl und führt für
                Henri zu einem halben Jahr Dunkelhaft. Doch er und seine Fluchtpartner wollen nicht
                aufgeben und organiseren erneut ein Boot.")
                .font(.body)
                .frame(maxWidth: .infinity, alignment: .leading)
                .padding(3)
            Text("IMDb: k.A. \n Genre: Krimi / Thriller \n Empfohlen von: PvD")
                .font(.body)
                .frame(maxWidth: .infinity, alignment: .leading)
                .padding(3)

        }

    }
}
```

Nachdem alles hübsch aussieht, kann man die Labels und das Image aktualisieren. Ab dieser Punkt entsprechen die Änderungen zu der File BroadcastRow.swift.

Damit wir die Variable innerhalb ProgramItem zugreifen können, brauchen wir erstmal eine variable der Typ ProgramItem. Hier habe ich diese Variable broadcast genannt.

Für die Labels kann man die entsprechenden variable über Broadcasts mit Punkt direkt zugreifen.

Für das Image habe ich das Initialisierer Image(uiImage:) verwendet. Diese Initialisierer nimmt eine UIImage als Parameter. Man kann einem UIImage mit dem Initialisierer UIImage(data:) mit einer URL als Parameter aufrufen. Weil diese Initialisierer einem optional zurückliefert, muss man sie mit dem Operator ! auspacken. Innerhalb der UIImage(data:) habe ich meiner Funktion myImage gerufen mit iconURL! String als Parameter. Den iconURL String ist auch eine optional und muss deshalb ausgepackt werden.

Merke, dass der BroadcastRow_Provider nicht mehr funktioniert. Das ist so, weil wir unsere Broadcast Daten noch herunterladen müssen. Am besten kann man alle Zeilen auf einmal selektieren und mit command + / auskommentieren.

```
struct BroadcastRow: View {

    var broadcast: ProgramItem

    var body: some View {
        VStack {
            Image(uiImage: UIImage(data: broadcast.myImage(iconUrlString: broadcast.iconURL!))!)
                .resizable()
                .aspectRatio(contentMode: .fit)
                .scaledToFit()
            Text(broadcast.title!)
                .font(.headline)
                .frame(maxWidth: .infinity, alignment: .leading)
                .padding(3)
            Text(broadcast.subtitle!)
                .font(.subheadline)
                .frame(maxWidth: .infinity, alignment: .leading)
                .padding(3)
            Text(broadcast.description!)
                .font(.body)
                .frame(maxWidth: .infinity, alignment: .leading)
                .padding(3)
            Text(broadcast.details!)
                .font(.body)
                .frame(maxWidth: .infinity, alignment: .leading)
                .padding(3)

        }
    }
}
```

3.5 Content View

Der Content View ist der Hauptview unsere App. hier werden alle andere Methoden aufgerufen.

Zu Anfangen braucht man hier die $@\mathrm{ObservedObject\ var\ programDownloader}$ der Typ $\mathrm{downloadProgram}$. Diese variable Beobachtet die $@\mathrm{Published\ var\ programItems}$ in der $\mathrm{ProgramItems}$ Struct. Ohne $@\mathrm{ObservedObject}$ würden die Änderungsankündigungen von der $@\mathrm{Published}$ Variable zwar gesendet, aber ignoriert [16].

In dem Initialisierer von $\mathrm{ContentView}$ wird diese Variable mit unserer URL initialisiert. Dafür benutzen wie unserer $\mathrm{downloadProgram.init(programURL:})$ Klasse. Danach wird das Unterprogramm $\mathrm{loadProgram}$ gerufen.

In der Body View möchten wir jetzt die Zellen einzeln aufrufen. Dafür verwenden wir eine dynamische liste [12] mit dem struct $\mathrm{forEach}$ [6]. Aus diesem Grund musste in der $\mathrm{ProgramItems}$ jeder Elemente das Protokoll Identifiable entsprechen. Eine ForEach-Instanz ruft das Array $\mathrm{programItems}$ mehrmals auf und erzeugt für jede item eine neue $\mathrm{BroadcastRow}$ Instanzen.

Auf die Liste wollte ich noch das Program Title haben. Deshalb habe ich ein Text mit dem String ”Kodinerds freeTV Announcement Service” und die Modifier .font(.title2) und .frame(maxWidth: .infinity, alignment: .leading).

Die liste und den Title werden zum Schluss in einem Vertical Stack eingebettet.

Somit ist das Implementierung mit dem SwiftUI Frameworks fertig!

4 Fazit

Nachdem Implementierung mittels Storyboard und Swift UI, habe ich festgestellt, dass die Implementierung mit SwiftUI mir besser gefällt. Zwar sieht das Interface Builder am Anfang sehr freundlich aus, aber sobald man anfängt die Outlets und Actions zu löschen oder umbenennen merkt man wie Fehler anfällig und nervig Storyboards sein können.

Zum beispiel: 1. Ein Connection wird mittels Control + Drag gemacht. 2. Wir löschen den code 3. Der Code zeigt kein Fehler, weil das Interface Builder egal ist, ob einer Verbindung zu einem code liefert. Das Interface Builder Objekte ist mit ”nichts” verbunden. Der code funktioniert aber der App läuft nicht wie erwartet.

Einem Storyboard Projekt kann auch sehr schnell sehr groß werden und man verliert dadurch den Überblick über Kleinigkeiten, wie Identifier Strings und die ganzen Connections.

Das liegt daran, dass der Interface-Builder keinem Swift Code generiert. Das Storyboard ist nur von Vorteil dann, falls man noch irgendeinem Code in den Alten Objektiv-C hat. Da den Interoperabilität zwischen Swift und Objective-C mit dem SwiftUI nicht funktioniert.

Der SwiftUI ist ein reines Swift-Framework. Die Befehle um einem Interface zu entwerfen ist sehr einfach und XCode macht alles noch einfacher durch dem Attributes Inspector und mit der Funktion von auf einem code Zeile command + klicken. Das Automatic Preview ist auch sehr hilfreich.

Quelle

[1] *App Development with Swift*. App Inc. - Education, 20 Nov, 2019, p. 618.

[2] *App Development with Swift*. App Inc. - Education, 20 Nov, 2019, pp. 890–892.

[3] Apple Developer Documentation. *Date*. url: https://developer.apple.com/documentation/foundation/date. (accessed: 03.06.2021).

[4] Apple Developer Documentation. *DateFormatter*. url: https://developer.apple.com/documentation/foundation/dateformatter. (accessed: 03.06.2021).

[5] Apple Developer Documentation. *downloadTask(with:completionHandler:)* url: https://developer.apple.com/documentation/foundation/urlsession/1411511-downloadtask. (accessed: 04.06.2021).

[6] Apple Developer Documentation. *ForEach*. url: https://developer.apple.com/documentation/swiftui/list. (accessed: 06.06.2021).

[7] Apple Developer Documentation. *IndexPath*. url: https://developer.apple.com/documentation/foundation/indexpath/. (accessed: 04.06.2021).

[8] Apple Developer Documentation. *init(contentsOf:options:)* url: https://developer.apple.com/documentation/foundation/data/3126626-init. (accessed: 04.06.2021).

[9] Apple Developer Documentation. *init(string:)* url: https://developer.apple.com/documentation/foundation/url/3126806-init. (accessed: 04.06.2021).

[10] Apple Developer Documentation. *jsonObject(with:options:)* url: https://developer.apple.com/documentation/foundation/jsonserialization/1415493-jsonobject. (accessed: 04.06.2021).

[11] Apple Developer Documentation. *JSONSerialization*. url: https://developer.apple.com/documentation/foundation/jsonserialization/. (accessed: 04.06.2021).

[12] Apple Developer Documentation. *List*. url: https://developer.apple.com/documentation/swiftui/foreach. (accessed: 06.06.2021).

[13] Apple Developer Documentation. *shared*. url: https://developer.apple.com/documentation/foundation/urlsession/1409000-shared. (accessed: 03.06.2021).

[14] Apple Developer Documentation. *URLSession*. url: https://developer.apple.com/documentation/foundation/urlsession. (accessed: 03.06.2021).

[15] Paul Hudson. *Hacking with Swift*. url: https://www.hackingwithswift.com/quick-start/swiftui/what-is-the-published-property-wrapper. (accessed: 06.06.2021).

[16] Paul Hudson. *Hacking with Swift*. url: https://www.hackingwithswift.com/quick-start/swiftui/what-is-the-observedobject-property-wrapper. (accessed: 06.06.2021).

[17] *Stackoverflow*. url: https://stackoverflow.com/questions/31254725/transport-security-has-blocked-a-cleartext-http. (accessed: 03.06.2021).