STUDIENARBEIT MODERNE SOFTWARE ARCHITEKTUR UNTER IOS

Erneuerung der News-App durch neues Architekturkonzept

Annika Brigitte Pfosch

Mobile Computing 5. Semester

Inhaltsverzeichnis

1. Vorwort

Die hier beschriebene App ist im Rahmen der Vorlesung „Moderne Softwarearchitektur unter iOS“ entstanden. Dies ist die Neuauflage einer Nachrichten-App, die ich bereits im vierten Fachsemester Mobile Computing realisiert habe. Die erste Nachrichten App ist mit Hilfe eines Model-View-Controller Patterns und mit UIKit, beziehungsweise dem Storyboard unter Xcode entwickelt worden. In der neueren Ausgabe wurde sie mit einer Model-View-ViewModel- Architektur aufgrund der neuen Frameworks SwiftUI und Combine umgesetzt. Im Folgenden werden Unterschiede und einige Probleme zwischen den Versionen dargestellt.

2. Systemspezifikation

Im folgenden Abschnitt wird die Funktionalität der Anwendung definiert und die Schnittstellen aus Sicht des Anwenders festgeschrieben.

2.1 Datenmodell

Die Anwendung soll vor Allem mit JSON-Daten von einer RESTful-Schnittstelle aus dem Internet arbeiten. Die Daten kommen von der Webseite https://newsapi.org. Die Website gibt JSON-Daten zurück, die in der App in passende Objekte der Klasse „Article“ umgewandelt werden.
Die Schnittstelle liefert immer ein Datenpaket mit 20 Artikeln unter dem Endpunkt: `https://newsapi.org/v2/top-headlines?[Land]&[Kategorie]&[API_Key]`.
Als Query-Parameter können können diverse Länder mit den entsprechenden Kürzeln angegeben werden. Somit erhält man nur Nachrichten aus einer bestimmten Sprache, zum Beispiel werden beim Kürzel „de“ nur deutsche Nachrichten zurückgegeben.
Die Nutzer können sich auch Nachrichten aus verschiedenen Kategorien wie Sport, Gesundheit oder Entertainment zurückgeben lassen. Am Ende einer Anfrage muss der individuelle API-Key angegeben werden. Damit authentifiziert man sich gegenüber der Schnittstelle und die Organisation kann die Anfrage zu einem bestimmten Nutzer mit einem Konto zuordnen.

2.2 Use-Case Diagramm

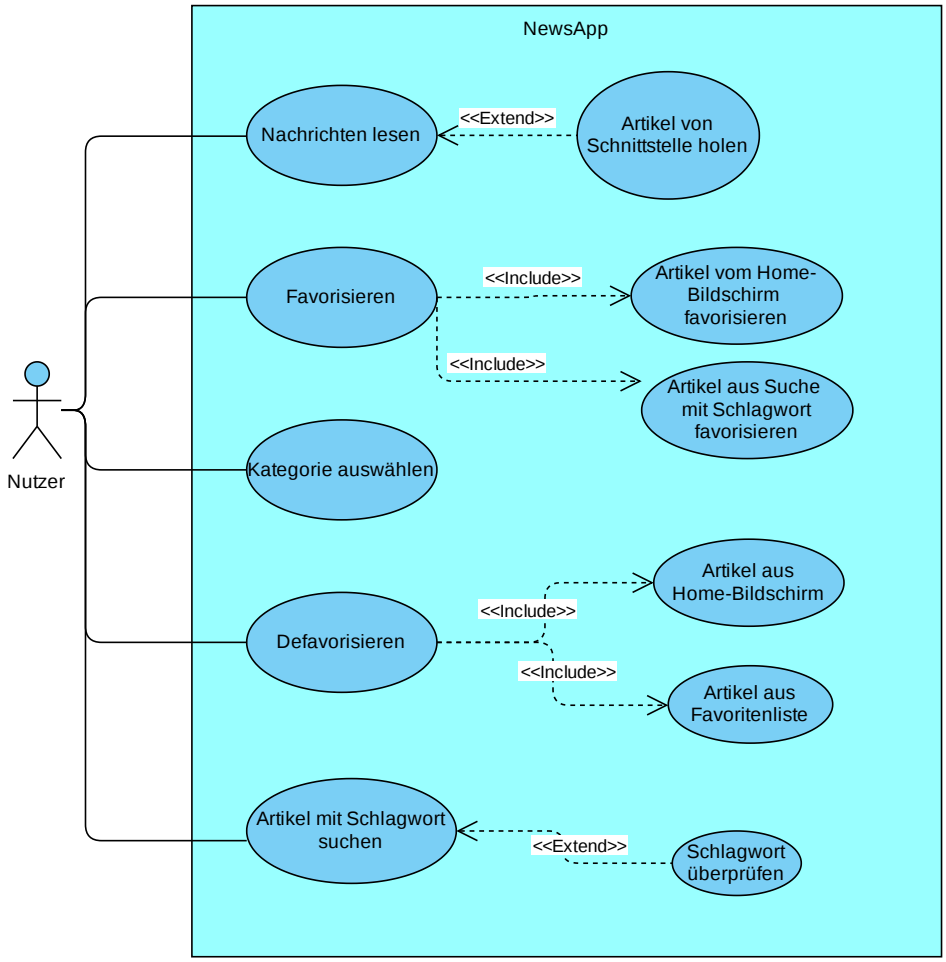


Abbildung 1: Use-Case Diagramm

In diesem Use-Case Diagramm werden die Anwendungsfälle eines Nutzers in der News-App dargestellt. Dies deckt sich auch mit den im Folgenden beschriebenen Funktionen.

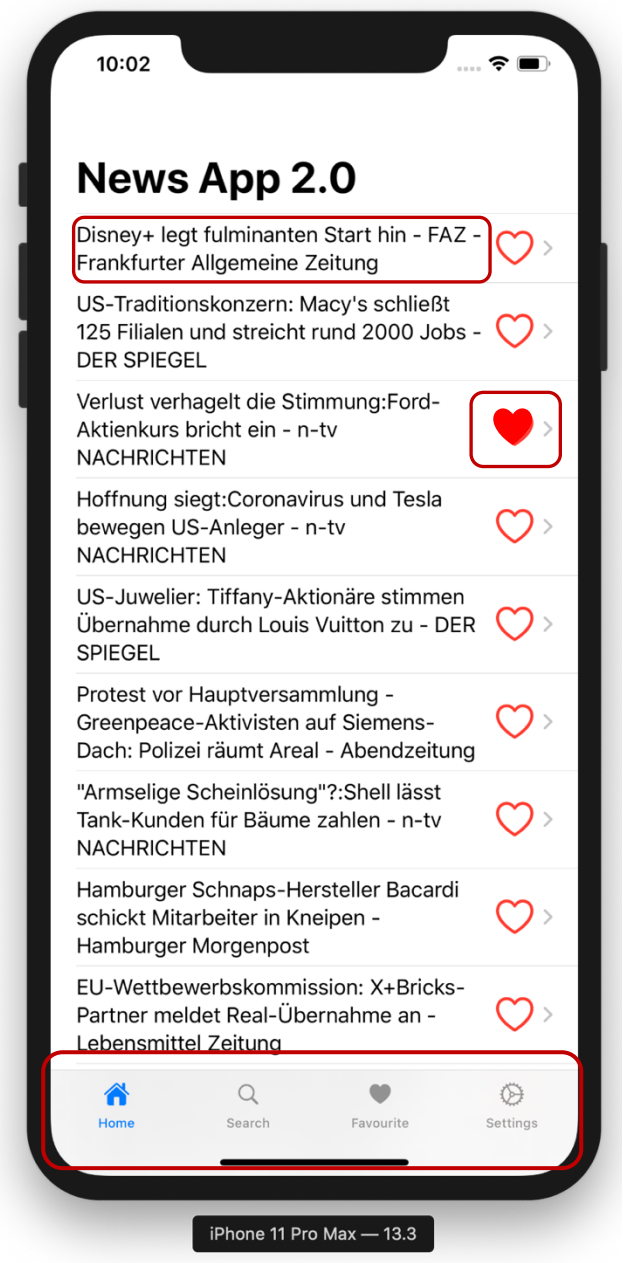
2.3 Funktionenmodell

Die Funktionen ändern sich gegenüber der alten Version der App nur geringfügig. Es ist weiterhin möglich, aktuelle Artikel auf dem Home-Bildschirm anzeigen zu lassen. Die Kategorie der Artikel kann in einem Einstellungsbildschirm festgelegt werden. Im Home-Bildschirm wird zunächst nur die Überschrift des Artikels und daneben ein Herz-Symbol angezeigt, mit dem es möglich ist, Artikel zu favorisieren. Sie werden dann in die Favoriten-Liste aufgenommen, die in einem anderen Bildschirm anzeigbar ist. Wenn der Nutzer auf die Überschrift eines Artikels im Home Bildschirm klickt, gelangt er zur Detailansicht des Artikels. In der Detailansicht ist dann der gesamte Artikel mit Bild zu sehen. Da newsapi.org den Artikel jedoch nicht in voller Länge zurückliefert, gibt es in der Detailansicht noch einen Button „mehr lesen“, der dann zu der Internetseite weiterleitet, auf der sich der Artikel in voller Länge zum Lesen befindet.
Die Favoriten werden persistiert, das heißt, dass auch nachdem die App vollständig beendet wurde, die Favoriten weiter gemerkt werden, bis der Nutzer dies nicht mehr möchte. Die

Artikel im Home-Bildschirm werden nicht persistent gespeichert, sie werden jedes Mal beim Start der App aktualisiert.

2.4 Benutzerschnittstelle

Die Daten werden durch verschiedene Bildschirme in einer iOS-App dargestellt.
Dabei ist ein Tab-View vorhanden, das heißt, dass es eine Navigationsleiste gibt, durch die man in verschiedene Bildschirme gelangt. Der Benutzer interagiert mit dem System über Touch und Änderungen des Benutzers werden sofort auf dem Bildschirm sichtbar:



Titel des Artikels: Weiterleitung zur Detailansicht bei Klick

Herz: Beim Klick auf das Herz wird dieser Artikel favorisiert und zur Favoritenliste hinzugefügt

Navigationsleiste: Home-Bildschirm, Suche, Favoriten und Einstellungen
Aktueller Tab wird durch farbliche Hervorhebung dargestellt

Abbildung 2: Home-Bildschirm

Da die Funktionalität im Gegensatz zur Version 1.0 der Nachrichten-App fast gleich ist, besteht ein Änderungbedarf der App nur wegen der neuen Softwarearchitektur, die mit SwiftUI und dem Combine-Framework einhergeht.

Als Basis für das Projekt dient Apples Programmiersprache Swift, sowie das Programm Xcode, das zur Erstellung eines Softwareprojekts für ein iPhone genutzt werden kann.

3. Systemkonstruktion

Im Folgenden Abschnitt wird beschrieben, wie die Software arbeitet und wie das System aufgeteilt ist. Dies geschieht durch Modularisierung, also der Aufteilung eines Systems in verschiedene Komponenten.

3.1 Das Model-View-Controller Pattern

Die vorherige Version dieser App wurde mit dem MVC-Pattern umgesetzt, welches wohl das weit verbreitetste Architekturmuster in Bezug auf App-Programmierung unter iOS ist.
Die Software wird hier in drei verschiedene Komponenten unterteilt: Der Präsentationsschicht (View), der Steuerungsschicht (Controller) und dem Datenmodell (Model).
Jede Schicht hat hier eine andere Aufgabe, sodass einzelne Module für eine spätere Programmierung wiederverwendet und leicht ausgetauscht werden können.
Das Model enthält alle Daten, die von den anderen Schichten zu einem Teil benötigt werden. Es ist für die persistente Speicherung und die Datenbereitstellung über alle Komponenten hinweg zuständig. Oftmals wird es durch ein sogenanntes Singleton-Muster implementiert, welches sicherstellt, dass es nur eine Instanz des Models gibt, das aber für alle Komponenten global zur Verfügung steht. Somit wird doppelte Datenhaltung verhindert und die Instanz wird erst erzeugt, sobald sie benötigt wird. Änderungen auf dem Datensatz sind somit leichter möglich.
Die Präsentationsschicht (der View) ist für die Darstellung der Daten und die Benutzerinteraktion zuständig. Sie ist von der Controlllerschicht abhängig. Demnach werden Benutzerinteraktionen an den Controller weitergegeben. Der View wird über Änderungen der Daten im Modell unterrichtet, wenn sich der View dafür beim Model anmeldet und kann daraufhin die Darstellung aktualisieren.
Die Steuerungsschicht kann mehrere Klassen, die als Controller fungieren, enthalten. Dabei kommuniziert der Controller mit der Präsentationsschicht und dem Model. Der Controller wird über Benutzerinteraktionen informiert, wertet diese aus und nimmt daraufhin Anpassungen an den Daten oder an der Darstellung vor.
In dieser Version des Projekts befand sich die Geschäftslogik in einzelnen Controller-Klassen, zum Beispiel die Kommunikation mit der Schnittstelle über das Internet oder die Umwandlung von Text in Sprache.

3.2 Architektur der Version 1.0 mit Model-View-Controller

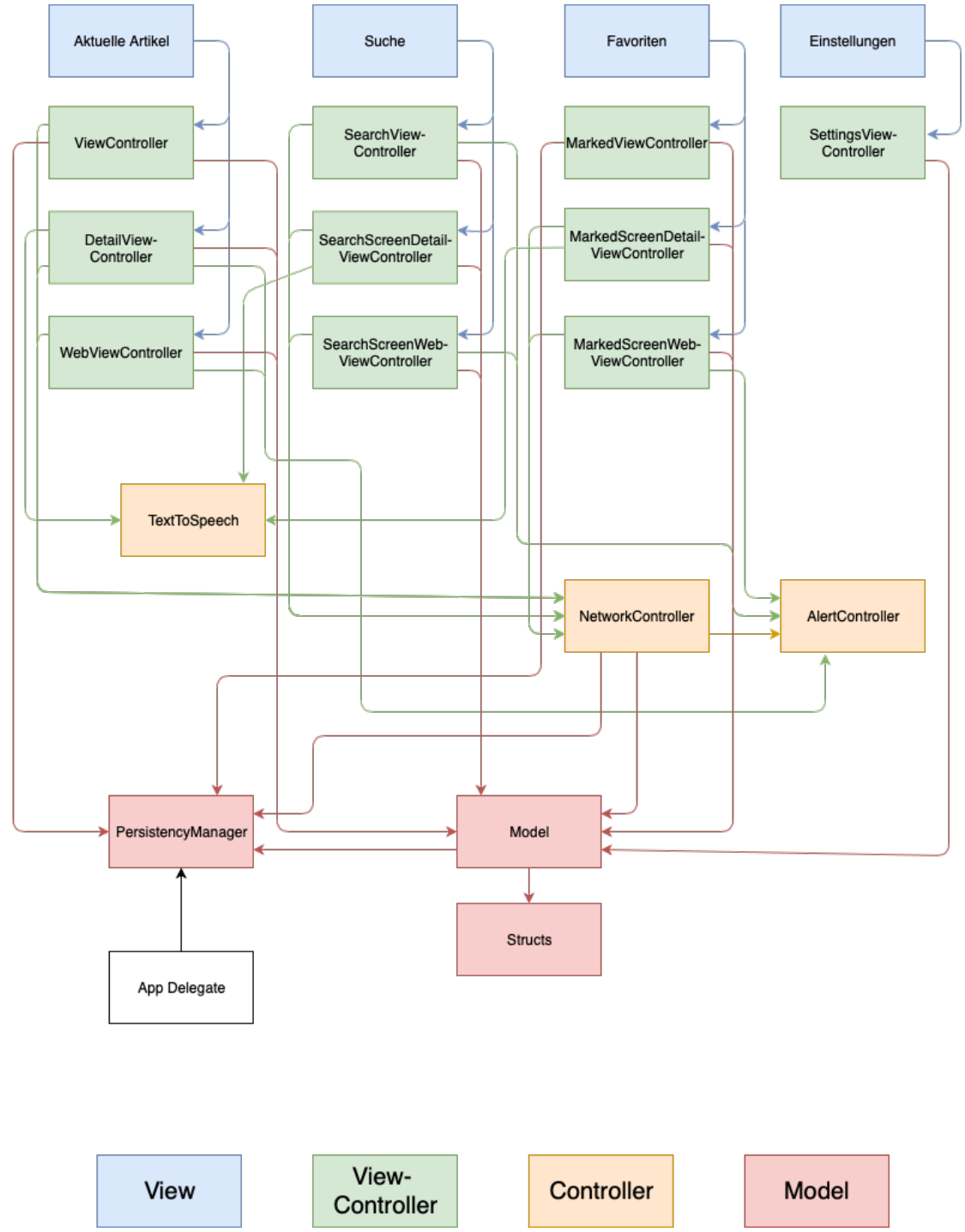


Abbildung 3: Architektur der alten Version

3.3 Das Model-View-Presenter oder Model-View-ViewModel - Pattern

Im Gegensatz zum M-V-C Modell werden die verschiedenen Komponenten mehr voneinander getrennt. In der Version 1.0 gab es ViewController, die nicht nur die Aufgabe eines Views übernahmen, sondern auch weniger komplizierte Aufgaben eines Controllers. Dies soll nun nicht mehr stattfinden. Die ViewController sind durch SwiftUI auch aus den Projekten verschwunden. Es gibt jetzt View-Files, die nur für die Präsentation zuständig sind.
In der MVVM-Architektur hat das ViewModel zwei verschiedene Aufgaben: Die Implementierung der Geschäftslogik und die Bereitstellung der relevanten Daten für den dazugehörigen View. Das ViewModel/ der Presenter hält also nur die Daten aus dem Model, die der View auch wirklich benötigt.
Ein ViewModel fungiert als „virtueller Sachbearbeiter“ und hat eine fest umschriebene Aufgabe. Es braucht einen definierten Input, um einen Output als Ergebnis bereitzustellen. Dabei können Teilaufgaben an andere ViewModels delegiert werden.
Ein View ist eine abgeschlossene graphische Oberfläche, welche keine Geschäftslogik enthält, sondern ausschließlich für die Darstellung der Daten aus dem ViewModel zuständig ist. Ein Bildschirm kann durch mehrere oder nur einen View dargestellt werden.
Durch SwiftUI und Combine ist eine hierarchische Sachbearbeiter-Struktur vorgegeben, die ViewModels sind die oberste Ebene eines Sachbearbeiters, weiter unten in der Struktur gibt es dann die Combine-Pipelines.[1]

3.4 Unterschiede und Gemeinsamkeiten

Bei der MVC-Architektur holt sich die View über das Beobachter-Muster die Daten direkt vom Model ab. Dadurch besteht eine gewisse Abhängigkeit zwischen Model und View.
Das MVVM-Muster sorgt dafür, dass die Daten, die die View braucht, nicht mehr direkt vom Model kommen, sondern vom dazugehörigen ViewModel. Im MVVM- Pattern synchronisieren sich View und ViewModel bidirektional, was in Swift und dem Combine-Framework durch die Property Wrapper `ObservedObject`, `Observable` und `Published` umgesetzt wird.
Somit fallen also auch Notifications weg, die im vorherigen Projekt genutzt wurden, um einzelne Komponenten über Änderungen zu Informieren.
In der neuen Architektur gibt es auch das Storyboard nicht mehr. Die Views werden jetzt programmatisch in einer Datei festgelegt. Das bietet bei Projekten auch den Vorteil, dass die Versionsverwaltung einfacher ist und die Entwickler sich besser an die Vorgaben der Designer halten können und umgekehrt.
Die Aufgabe des Models ist in diesem Modell gleichgeblieben, die Model-Instanz und der Persistenz-Manager konnten bis auf ein paar kleine Änderungen wiederverwendet werden.

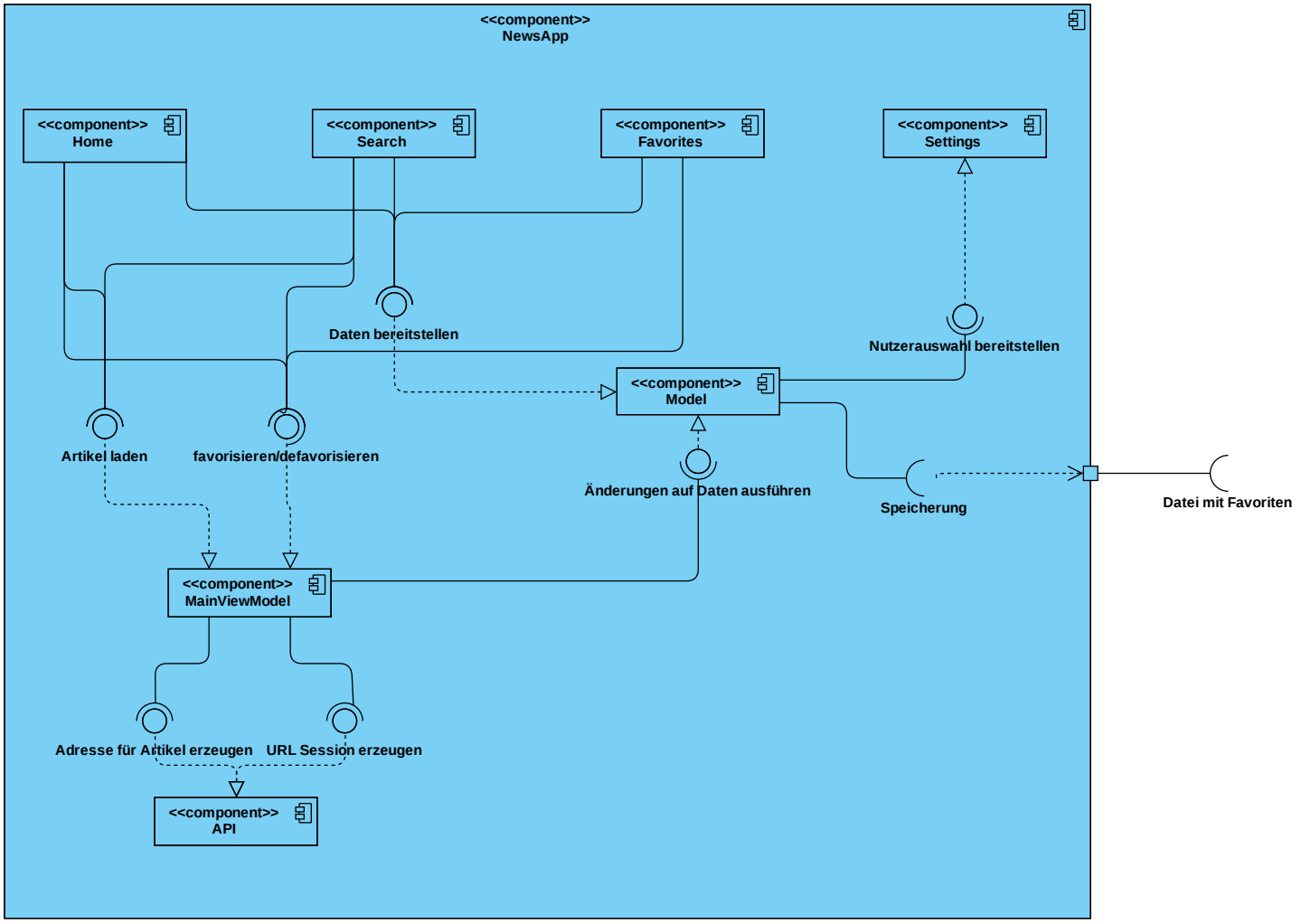
3.5 Komponentendiagramm

Das folgende Komponentendiagramm beschreibt, in welche Komponenten die Nachrichten-App aufgeteilt ist. Anschließend werden die Komponenten im Detail aufgezeigt, das heißt, welche Komponenten welche Module besitzen.

---

[1] Einige Aspekte übernommen aus: Skript zur Vorlesung Moderne Software Architekturunter iOS, Prof. Dr. Peter Stöhr, Hochschule Hof, Wintersemester 2019/2020

Die große Komponente `NewsApp` besteht aus einzelnen Komponenten, die zum einen die verschiedenen Bildschirme (also Home, Suche, Favoriten und Einstellungen) widerspiegeln, zum anderen aber gewissen Funktionen haben, wie die Model oder API- Komponente.
Die Komponente `MainViewModel` ist im Projekt besonders, da sie für die drei Views von Home, Suche und Favoriten die Funktion des Favorisierens/Defavorisierens und die Funktion zur Weiterleitung auf die Detailseite besitzt.
In der vorherigen Version musste pro Bildschirm diese Funktion immer in einer seperaten Klasse erfolgen.



Dies sehe ich als großen Vorteil gegenüber dem Storyboard: Durch SwiftUI kann man Views und die dazugehörigen ViewModels in verschiedenen Bildschirmen wiederverwenden. Im Storyboard ist dies nicht möglich.

Einige Module sind hier in der Übersicht nicht aufgeführt, wie zum Beispiel die `WebView` oder die `DetailView`. Diese werden im Folgenden natürlich genauer erklärt, jedoch sind sie nicht so groß, um diese in einem Komponentendiagramm aufzunehmen, da sie die Übersicht nur komplizierter machen würden. Deswegen wurden sie hier bewusst weggelassen.

Abbildung 4: Komponentendiagramm

3.6 Aufteilung in Module

Nun werden die einzelnen Komponenten genauer betrachtet. Jede Komponente besteht aus einzelnen Modulen, welche als Swift-Dateien im Programmcode repräsentiert sind.

3.6.1 News App 1.0

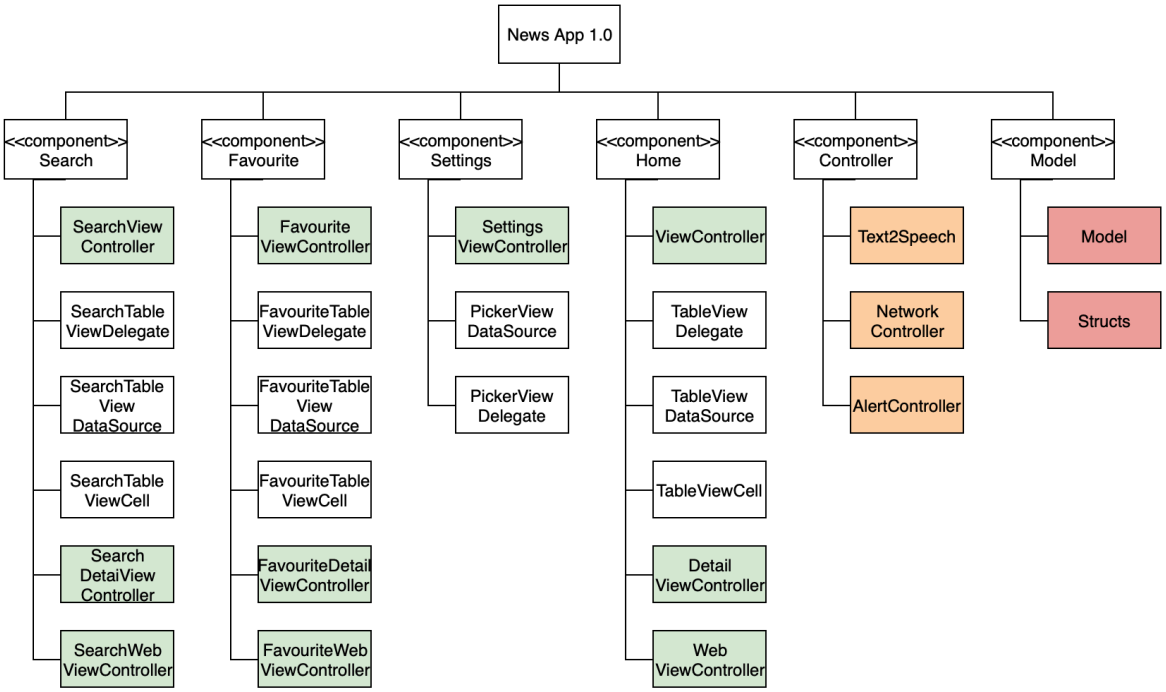


Abbildung 5: Strukturdiagramm Version 1.0

3.6.2 News App 2.0

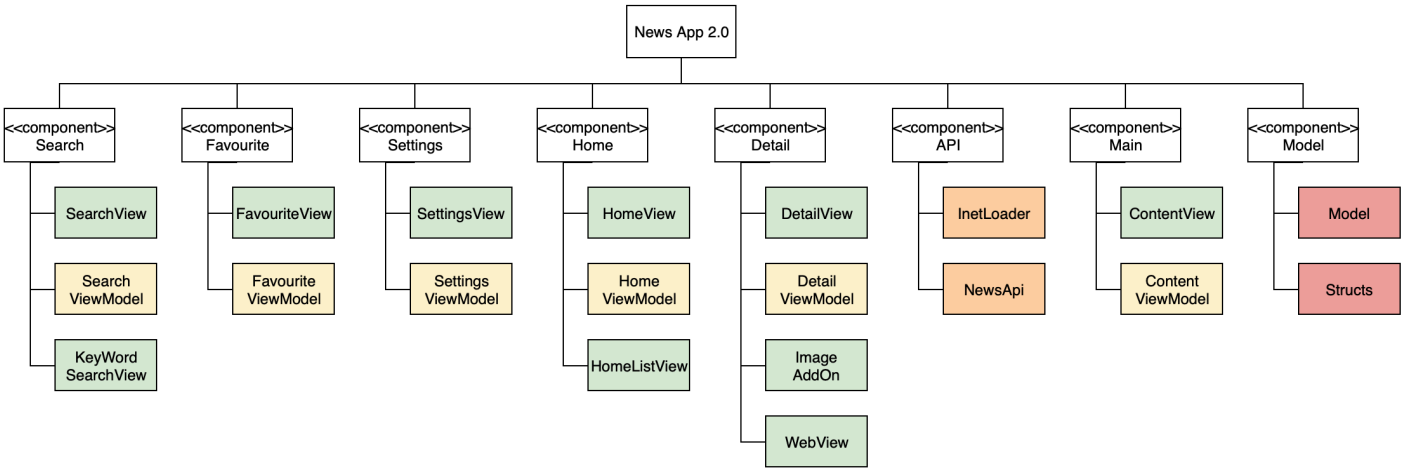


Abbildung 6: Strukturdiagramm Version 2.0

3.6.3 Vergleich

Die erste Version der App ist in weniger Module aufgeteilt. Im Gegensatz zur neueren Version müssen in den Komponenten Search, Favourite und Home immer wieder `WebViewController` und `DetailViewController` erstellt werden. Dies fällt in der neueren Version weg, da jede Komponente auf den `WebView` und den `DetailView` zugreifen kann, da diese Views jetzt Programmcode sind und nicht mehr im Storyboard verankert sind. Die dazugehörigen ViewModels können durch eine @Observed Annotation mit eingebunden werden. Dies ist ein wesentlicher Vorteil gegenüber der alten Version, da so doppelter Code vermieden wird und die Views und ViewModels automatisch über Änderungen informiert werden. Dies musste der Entwickler vorher selbst durch Notifications durchführen.
In der zweiten Version ist die Model-View-ViewModel Architektur auch durch die Struktur der Komponenten ersichtlich: Eine Komponente besteht demnach aus Views und einem zugehörigen ViewModel. Das Model-Singleton ist eine extra Komponente, und kann von allen ViewModels verwendet werden. Die Komponente API ist hier besonders: Hinter dieser befindet sich die Geschäftslogik für das Holen der Daten aus dem Internet und den Aufbau einer URL.
Die Komponenten Home, Settings, Favourite und Search werden in der Komponente Main erzeugt. Die Main-Komponente ist der Startbildschirm, den die Nutzer sehen, mit den verschiedenen Bildschirmen, durch die der Nutzer mit der Navigationsleiste navigieren kann. Die Komponente API wird vom Modul `MainViewModel` verwendet, um eine jeweilige URL zu erstellen und die Daten zu holen. Das Modul MainViewModel wird in den Komponenten Search, Favourite, Home und Settings verwendet, um zum Beispiel das Favorisieren zu ermöglichen, denn die Logik des Favorisierens/Defavorisierens ist in diesem Modul enthalten.

4. Modulprogrammierung

4.1 Main

4.1.1 ContentView

Im `ContentView` wird das Grundgerüst der Präsentation dargestellt. Dieser View enthält die jeweils anderen Views `HomeView, SearchView, FavouriteView` und `SettingsView` mit ihren jeweiligen ViewModels. Diese sind alle in einer TabBar verankert, sodass der Nutzer durch die verschiedenen Screens navigieren kann.

4.1.2 ContentViewModel

Diese Klasse stellt ein ViewModel dar. Es lädt bei Bedarf Daten aus dem Internet. Diese Aufgabe delegiert dieses Model an die Klasse `InetLoader`. Wenn der Nutzer im `HomeView`, im `SearchView` oder im `FavouriteView` nun auf das Herzchen neben dem Artikel klickt, ist hier die Logik implementiert, diesen Artikel in die Favoritenliste aufzunehmen. Das Herzchen wird nun im `HomeListView` ausgefüllt. Beim erneuten Klick auf das Herzchen ist es nun nicht mehr ausgefüllt und der Artikel wird aus der Favoritenliste entfernt.

4.2 Home

4.2.1 HomeView

Dieses Modul repräsentiert den Home-Bildschirm. In ihm ist eine Tabelle mit mehreren `HomeListViews` enthalten. Die HomeListViews entsprechen den Zellen der Tabelle. Diese füllt sich dynamisch mit dem Inhalt des DataSources.

4.2.2 HomeListView

Dieser View entspricht einer Zelle einer Tabelle. Sie besteht aus dem Titel des Artikels und einem Herzchen als Symbol daneben.

4.3 Search

4.3.1 SearchView

Dieser View beinhaltet die Darstellung eines Textfeldes zur Benutzereingabe mit einem Button und einer Tabelle. Wenn der Nutzer ein Suchwort in das Textfeld eingibt und auf „Los geht's" klickt, werden in der Tabelle alle relevanten Suchergebnisse dargestellt.

4.3.2 KeyWordSearchView

Dieser View ist ein kleiner Teil des Hauptviews `SearchView`. Hier ist das Textfeld zusammen mit dem Button implementiert. Dieses Fragment kann dann in mehreren anderen Views eingesetzt werden.

4.3.3 SearchViewModel

Diese Klasse ist ein ViewModel und stellt die Daten, die der `SearchView` vom Model braucht, in einem `DataSource` zu Verfügung. Dieser DataSource kann nun von einem View angesprochen werden. In der Klasse ist eine `@Published` Variable `allDone` enthalten, die auf `true` gesetzt wird, wenn die Artikel, die das eingegebene Schlagwort enthalten, geladen wurden und der DataSource aufgebaut wurde. Der `SearchView` implementiert das `SearchViewModel` dann mit der Annotation `@ObservedObject`. Damit findet die Bindung zwischen dem View und dem ViewModel statt und der View wird informiert, dass sich etwas im DataSource geändert hat. Dadurch lädt sich der View neu und die neue Information kann dargestellt werden, ohne eine Notification verwendet zu haben.

4.4 Favourite

4.4.1 FavouriteView

In diesem View werden die favorisierten Artikel angezeigt.

4.4.2 FavouriteViewModel

Diese Klasse stellt ein ViewModel dar und holt sich vom Model alle relevanten Daten, nämlich nur die Artikel, die als Favorit gespeichert sind und packt sie in eine Datenquelle. Auch hier gibt es wieder eine Observed-Variable `allDone`, die den View über Änderungen informiert.

4.5 Settings

4.5.1 SettingsView

Im `SettingsView` kann der Nutzer diverse Einstellungen vornehmen. Bisher wurde aber nur implementiert, dass sich der Nutzer zwischen verschiedenen Kategorien entscheiden kann, aus denen die Liste der Artikel im Home-Bildschirm besteht. Diese wird bei Änderung in den UserDefaults gespeichert.

4.5.2 PickerView

In diesem View kann durch die verschiedenen Kategorien mit einem PickerView durchgescrollt werden. Dieser Unterview wird dann im `SettingsView` verwendet, um die Kategorien anzuzeigen.

4.5.3 SettingsViewModel

Im `SettingsViewModel` verbirgt sich die Logik des `SettingsViews`. Bisher wird hier nur eine Datenquelle mit den im Model vorhandenen Kategorien erstellt.

4.6 Detail

4.6.1 DetailView

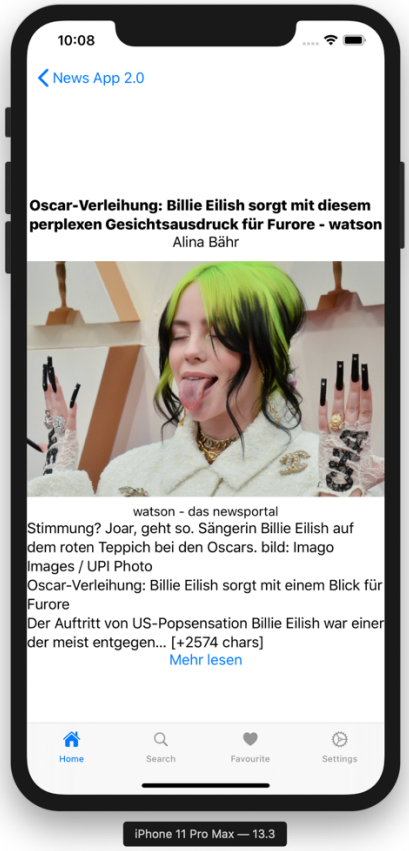


Abbildung 7: Detail-View

Im `DetailView` wird nun der gesamte Artikel mit Bild angezeigt, der von der Schnittstelle kommt. Wenn der Nutzer auf den Button „Mehr lesen“ klickt, erscheint die `WebView`, die auf die Seite mit dem vollständigen Artikel weiterleitet.

4.6.2 ImageAddOn

Dieses Modul ist für das Laden des Bildes zuständig. Für jeden Artikel gibt es eine URL `urlToImage`, die hier verwendet wird. Das Laden findet asynchron statt.

4.6.3 DetailViewModel

Hier wird das Foto zunächst mit einem Standard-Symbol initialisiert, solange dieses noch nicht geladen hat. Hier wird auch die URL zum Foto generiert, welche dann vom `ImageAddOn` verwendet wird.

4.6.4 WebView

Beim Klick auf „Mehr lesen“ wird der Nutzer über einen Navigations-Link in die `WebView` weitergeleitet. Hier befindet sich nun der gesamte Artikel.
Da SwiftUI noch kein Display-Element `WebView` besitzt, wird die `WKWebView` von UIKit als `UIViewRepresentable` implementiert. Das heißt, `WebView` ist eine View, die eine UIView repräsentiert.

4.7 API

Dieses Modul ist für das Holen der Daten aus dem Internet zuständig. Dies funktioniert nun anders als vorher. Die Sequenzdiagramme für das Laden aus dem Internet zeigen dies deutlich:

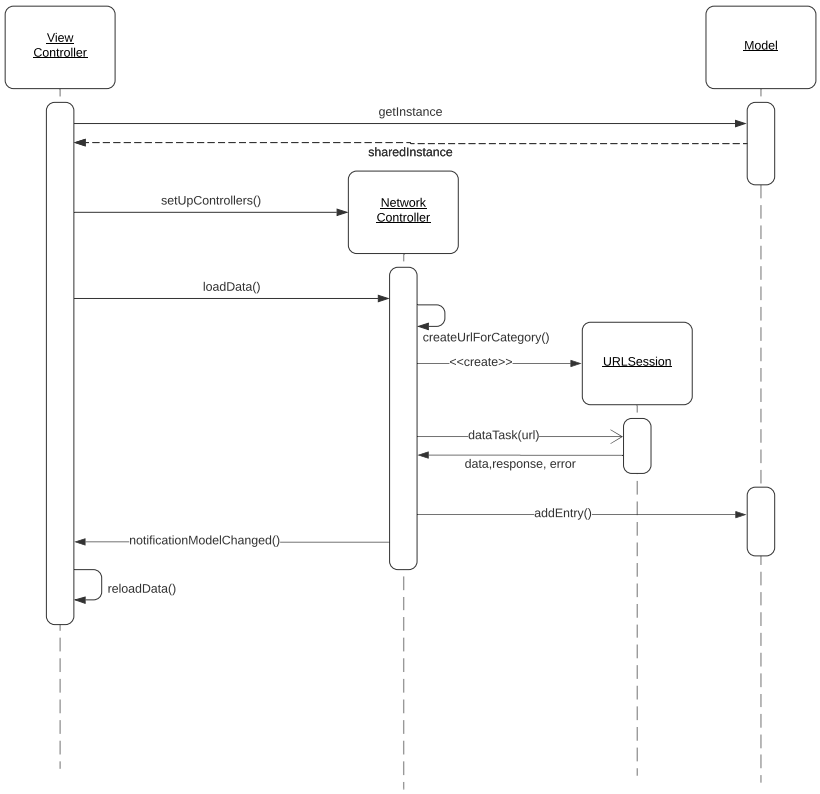


Abbildung 8: Sequenzdiagramm Version 1.0

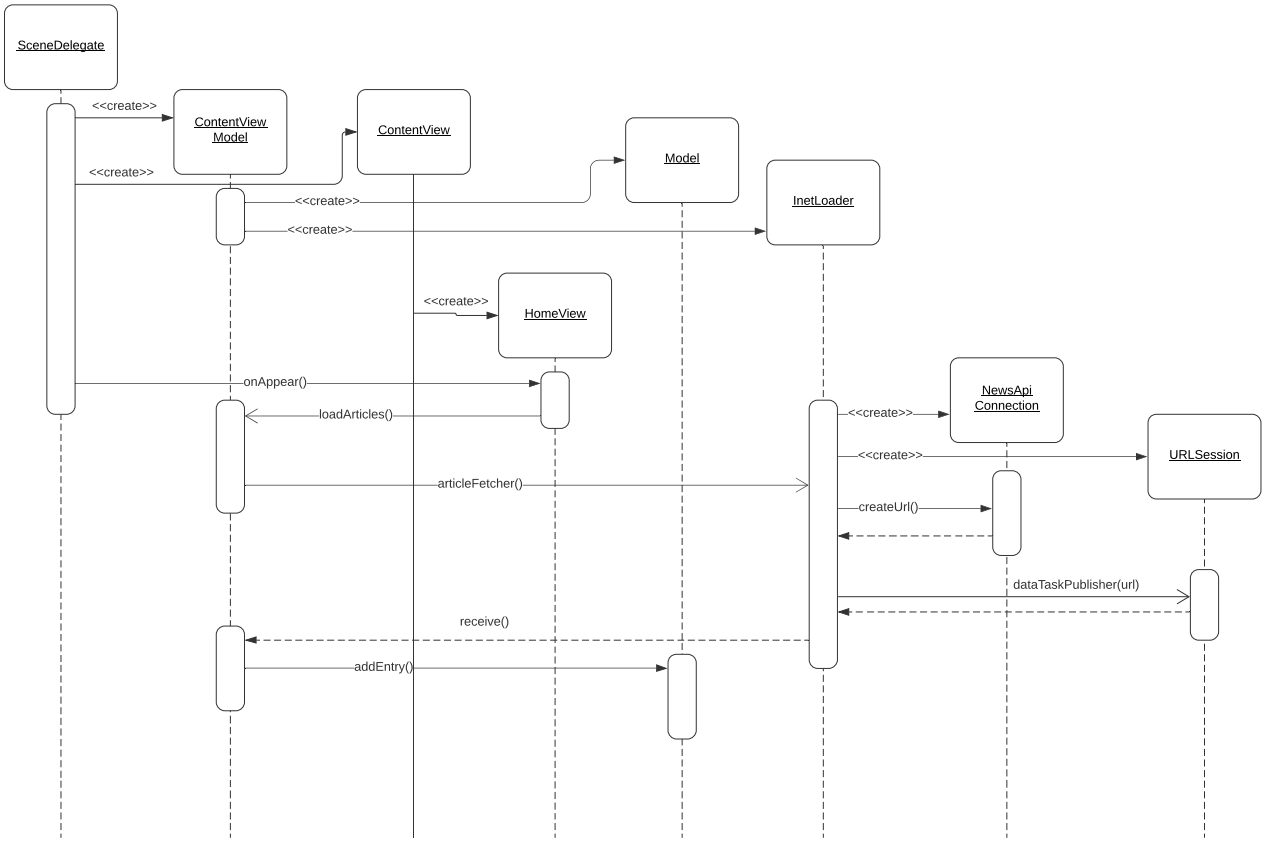


Abbildung 9: Sequenzdiagramm Version 2.0

Im Vergleich ist zu erkennen, dass nun deutlich mehr Module zum Einsatz kommen. Die `URLSession` wird jedoch weiterhin verwendet, um die Daten aus dem Netz zu laden. In der voherigen Version waren die Methoden für das Kreieren der URLs und das tatsächliche Laden in einer Klasse. Dies wurde nun getrennt. In der neuen Version finden durch die Combine-Pipelines mehrere asynchrone Aufrufe statt, die im Hintergrund die Daten laden.

4.7.1 InetLoader

In dieser Klasse wird eine andere Klasse `URLSession` verwendet, die es unter anderem ermöglicht, Daten von der API herunterzuladen. Dazu braucht `URLSession` eine URL, die sie von dem struct `NewsApiConnection` bekommt. Diese Klasse ist ein Beispiel für das Sachbearbeiter-Modell, das Apple vorgesehen hat. In dieser Klasse wurde das Combine-Framework mit Bearbeitungs-Pipelines verwendet. Eine Bearbeitungs-Pipeline besteht aus einem Publisher, einem Subscriber und Operatoren dazwischen, die die Daten modifizieren können. Die 2 Methoden der Klasse `InetLoader` geben jeweils einen Publisher zurück. Der Publisher veröffentlicht die Daten nur an den Subscriber, wenn der Task abgeschlossen ist und die Daten da sind. Wenn etwas nicht funktioniert hat, wird ein Fehler weitergegeben. Im `ContentViewModel` befindet sich der Subscriber, der die Daten dann vom Publisher erhält. Der Subscriber ist als `AnyCancellable` implementiert, das heißt, der Publisher und damit das Laden der Daten kann bei Bedarf unterbrochen werden.

4.7.2 NewsApi

In diesem struct befindet sich die Logik, die URLs für einige Endpunkte der Schnittstelle aufzubauen. Hierzu gibt es zwei Methoden: Die URL entweder anhand der vom Benutzer eingegebenen Kategorie aufzubauen, oder die URL durch ein Schlagwort, das vom Benutzer im SearchView eingegeben wurde aufzubauen.

4.8 Model

4.8.1 Model

Im Model befindet sich die gesamte Datenhaltung und Methoden, um auf diese Daten zuzugreifen und gegebenenfalls zu verändern. Das Model ist mit dem oben beschriebenen Singleton-Pattern implementiert. Zusätzlich zur `fileprivate class Model` gibt es die Klasse `PersistencyManager`, die für die Speicherung der Daten auf dem Smartphone zuständig ist.

4.8.2 Structs

Diese Datei hält die structs `ResponseObject`, `Article` und `Source`. Die Daten, die von der Schnittstelle kommen, werden dann in diese structs umgewandelt. Dabei kommt von der Schnittstelle ein Bündel mit mehreren Artikeln zurück, welches hier als `ResponseObject` bezeichnet wird. Ein Artikel wird durch `Article` repräsentiert. Jeder Artikel enthält eine Quelle, hier als `Source` bezeichnet.
Im folgenden Bild werden die einzelnen JSON-Daten beschrieben, die von der API kommen :

Response object

status	string	If the request was successful or not. Options: `ok` , `error` . In the case of `error` a `code` and `message` property will be populated.
totalResults	int	The total number of results available for your request.
articles	array[article]	The results of the request.
source	object	The identifier `id` and a display name `name` for the source this article came from.
author	string	The author of the article
title	string	The headline or title of the article.
description	string	A description or snippet from the article.
url	string	The direct URL to the article.
urlToImage	string	The URL to a relevant image for the article.
publishedAt	string	The date and time that the article was published, in UTC (+000)
content	string	The unformatted content of the article, where available. This is truncated to 260 chars for Developer plan users.

Abbildung 10: Key-Value Paare der Schnittstelle

In diese Swift structs werden die JSON-Objekte der Schnittstelle dann umgewandelt:

```
public struct ResponseObject : Codable {
    var status : String?
    var totalResults: Int?
    var articles : [Article]
}

public struct Article : Codable, Identifiable {
    public var id: UUID
    var source: Source?
    var author: String?
    var title : String?
    var description : String?
    var url : String?
    var urlToImage: String?
    var publishedAt : String?
    var content : String?
    var favorite : Bool?
```

Abbildung 11: structs

5. Abschließendes Fazit

Ein großer Vorteil von SwiftUI ist, dass die Views nun programmatisch erstellt werden, denn die Views können somit wiederverwendet werden. Nach einer „Eingewöhnunsphase“ ist der View leichter aufzubauen, denn der View wird schon von Anfang an so dargestellt, dass er die Daten angemessen anzeigt, und nicht erst Constraints verwendet werden müssen. In beiden Architekturen konnten die Structs und das komplette Model gleichermaßen eingesetzt werden, somit kann das Model wiederverwendet werden und die Austauschbarkeit von Modulen ist gegeben.
Ein weiterer Vorteil von SwiftUI mit dem Combine Framework ist die Publish-Subscribe-Architektur, die Notifications werden dadurch nicht mehr benötigt, um die Module über Änderungen zu informieren.
Neben den vielen Vorteilen ergaben sich für mich aber auch einige Probleme. Seit Beginn meines Studiums wurde mir beigebracht, die Model-View-Controller-Architektur zu programmieren. Demnach fiel es mir sehr schwer, auf eine andere Architektur umzusteigen und mich daran zu gewöhnen. Auch die „Single Source of Truth“ von Apple bereitete mir Schwierigkeiten, da es ja ein neues Konzept mit den ViewModels ist, die zwar auch Daten wie das Model halten, aber eben nicht alle, sondern nur die benötigten. Nach wie vor fällt es mir auch schwer, die Combine-Pipelines nachzuvollziehen und Operatoren und Pipelines an sich sinnvoll einzusetzen. Ein weiterer Nachteil an Apples neuen Frameworks ist, dass es wenig Dokumentation dazu gibt. Die einzigen Quellen sind eigentlich die WWDC 2019 Videos, die die Neuerungen versuchen zu erklären.

6. Abbildungsverzeichnis

7. Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne unerlaubte fremde Hilfe angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die den verwendeten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.