

Universität Heidelberg

Sommersemester 2016

Softwareentwicklung für iOS

App Katalog

Nils Fischer

Aktualisiert am 5. Juni 2016

Kursdetails und begleitende Materialien auf der Vorlesungswebseite:
<http://ios-dev-kurs.github.io>

Inhaltsverzeichnis

1 Einleitung	3
1.1 Über dieses Dokument	3
1.2 Workflow mit Git	3
1.2.1 Ein Repository forken, klonen und bearbeiten	4
1.2.2 Eine Aufgabe per Pull-Request einreichen	5
2 Hello World	6
2.1 "Hello World!" auf Simulator und Gerät	6
2.2 Graphisches "Hello World!"	7
Übungsaufgabe: Simple UI	13
3 A Swift Tour	16
Übungsaufgabe: Optional: Fibonacci	17
Übungsaufgabe: Optional: Primzahlen	17
Übungsaufgabe: Chatter	17
Übungsaufgabe: Poker	19
4 iOS App Architektur	22
Übungsaufgabe: Lifetime	23
Übungsaufgabe: Seasonizer	26
5 Methoden	29
5.1 Kommunikation mit einer REST API	29
Übungsaufgabe: API Client	30
5.2 Unit Tests	35
Übungsaufgabe: Unit Tests	36
5.3 Data Persistence	42
5.3.1 User Defaults	42
5.3.2 Caches	43
5.3.3 State Preservation	43
5.3.4 Datenbanken: Core Data und Realm	43
Übungsaufgabe: Data Persistence	46

Kapitel 1

Einleitung

1.1 Über dieses Dokument

Dieser App Katalog enthält Schritt-für-Schritt Anleitungen für die im Rahmen unseres Kurses erstellten Apps sowie die wöchentlich zu bearbeitenden Übungsaufgaben und wird im Verlauf des Semesters kapitelweise auf der Vorlesungswebseite [¹] zur Verfügung gestellt.

Er dient jedoch nur als Ergänzung zum parallel verfügbaren **Skript**, auf das hier häufig verwiesen wird. Dort sind die Erläuterungen zu den verwendeten Technologien, Methoden und Begriffen zu finden.

1.2 Workflow mit Git

Wir arbeiten in diesem Kurs mit der Versionskontroll-Software Git und der Softwareentwicklungs-Plattform GitHub [²]. Mit diesen Werkzeugen kann ich euch Beispielprojekte und Aufgaben bereitstellen, die ihr bearbeiten und mir für Kommentare wieder zur Verfügung stellen könnt. Gleichzeitig lernt ihr dabei direkt den Umgang mit zwei der wichtigsten Werkzeuge in der modernen Softwareentwicklung.

Mit Git können wir Änderungen an einem Projekt, oder *Repository*, in regelmäßigen Abständen in *Commits* speichern. Dann können wir jederzeit zu vorherigen Commits zurückkehren und Änderungen vergleichen. Wer die Speicherpunkte bei Super Mario kennt weiß so etwas zu schätzen.

Außerdem ermöglicht uns Git mit anderen Entwicklern zusammenzuarbeiten. Dazu können wir das Repository auf einem Server wie GitHub bereitstellen. Speichert ein anderer Entwickler Commits in dem Repository, können wir dessen Änderungen mit einem *Merge* mit unseren zusammenführen und dabei gegebenenfalls Konflikte beheben. So wird an Softwareprojekten weltweit zusammengearbeitet.

¹<http://ios-dev-kurs.github.io/>

²<https://github.com/>

Da Git ein Kommandozeilenprogramm ist, bedarf es sicherlich einer Eingewöhnung. Wenn ihr noch wenig Erfahrung im Umgang mit der Kommandozeile habt könnt ihr zum Einstieg die GitHub Desktop App [³] verwenden, mit der ihr Git über eine graphische Oberfläche bedienen könnt.

1.2.1 Ein Repository forken, klonen und bearbeiten

Ich stelle Beispielprojekte und Aufgaben in Repositories wie diesem [⁴] bereit. Verfahrt wie folgt, um es zum Bearbeiten herunterzuladen:

1. Erstellt einen GitHub Account [⁵], wenn ihr noch keinen habt. Ladet euch die GitHub Desktop App herunter, wenn ihr eine graphische Oberfläche der Kommandozeile vorzieht.
2. Ihr habt nur Lese-Zugriff auf mein Repository. Ihr müsst daher erst einen *Fork* [⁶] des Repositories erstellen und es damit auf euren Account kopieren. Klickt dazu einfach auf den *Fork* Button auf der Repository-Seite.
3. Euren Fork könnt ihr nun nach Belieben bearbeiten. In diesem Beispiel ist das Fork-Repository unter der URL <https://github.com/dein-username/helloworld> verfügbar. Ihr könnt die Kommandozeile oder die GitHub Desktop App verwenden um das Repository herunterzuladen, oder zu *klonen* [⁷]. Im Terminal lautet der Befehl dazu:

```
1 git clone https://github.com/dein-username/helloworld
```

4. Nun könnt ihr an dem Projekt arbeiten. Mit folgendem Befehl könnt ihr jederzeit überprüfen, welche Dateien sich geändert haben:

```
1 git status
```

5. Speichert in regelmäßigen Abständen *Commits* [⁸]. Jeder Arbeitsschritt sollte durch einen Commit repräsentiert werden. Achtet darauf, dass das Projekt bei jedem Commit funktionsfähig ist. In der Kommandozeile erstellt ihr einen Commit wie folgt:

```
1 # Status überprüfen
2 git status
3 # Alle Änderungen dem nächsten Commit hinzufügen
```

³<https://desktop.github.com>

⁴<https://github.com/ios-dev-kurs/helloworld>

⁵<https://github.com/join>

⁶<https://guides.github.com/activities/forking/>

⁷<http://gitref.org/creating/#clone>

⁸<http://gitref.org/basic/#commit>

```
4 git add --all  
5 # Commit durchführen  
6 git commit -m "Kurze Beschreibung der Änderungen"
```

6. Ihr könnt euer lokales Repository jederzeit mit eurem Repository auf GitHub abgleichen. Daher könnt ihr auch problemlos auf verschiedenen Rechnern an einem Projekt arbeiten. Führt einen *Push* [9] oder *Pull* [10] in der Kommandozeile aus, oder klickt den *Sync* Button in der GitHub Desktop App:

```
1 # Fortschritt auf GitHub veröffentlichen  
2 git push  
3 # Änderungen von GitHub herunterladen  
4 git pull
```

1.2.2 Eine Aufgabe per Pull-Request einreichen

Habt ihr eine Aufgabe fertig und möchtet Sie einreichen, oder wenn ihr Hilfe benötigt, erstellt eine *Pull-Request* [11]. Damit erhalte ich eine Benachrichtigung mit den Änderungen eures Forks im Vergleich zu meinem Original-Repository. Geht wie folgt vor:

1. Speichert eure Änderungen in einem Commit und veröffentlicht sie auf GitHub, wenn ihr es noch nicht getan habt.
2. Klickt auf der Repository-Seit den Button *New Pull Request*, überprüft die Änderungen und klickt dann *Create Pull Request*.
3. Gebt der Pull-Request einen Titel und beschreibt kurz die Änderungen. Erwähnt, wenn etwas nicht funktioniert, sodass ich euch helfen kann. Klickt schließlich auf *Create Pull Request*.

⁹<http://gitref.org/remotes/#push>

¹⁰<http://gitref.org/remotes/#pull>

¹¹<https://help.github.com/articles/creating-a-pull-request/>

Kapitel 2

Hello World

Was ist schon ein Programmierkurs, der nicht mit einem klassischen *Hello World* Programm beginnt? Wir werden jedoch noch einen Schritt weitergehen und diesen Gruß graphisch vom iOS Simulator und, soweit vorhanden, direkt von unseren eigenen iOS Geräten ausgeben lassen. Dabei stoßen wir auf unseren ersten *Swift* Code und lernen die IDE *Xcode* kennen. Wir arbeiten außerdem direkt mit der Versionskontroll-Software *Git*, einem der Grundbausteine nahezu jedes Softwareprojekts.

Relevante Kapitel im Skript: Xcode, Programmieren in Swift, Versionskontrolle mit Git sowie das Buch The Swift Programming Language [1]

2.1 "Hello World!" auf Simulator und Gerät

1. Ich habe ein Beispielprojekt bereitgestellt, anhand dessen wir einen ersten Blick auf die Programmierung einer iOS App werfen. Das Ziel ist, das Projekt herunterzuladen, eine erste, einfache App zu schreiben und mir das Ergebnis für etwas *konstruktive Kritik* zur Verfügung zu stellen. Dazu verwenden wir die Versionskontroll-Software *Git* und die Softwareentwicklungs-Plattform *GitHub*, die zu den Werkzeugen gehören, auf denen Softwareprojekte weltweit aufbauen und ohne die moderne Programmierung kaum noch denkbar ist.

Die erste Anweisung lautet:

Klont einen Fork des Repositories <https://github.com/ios-dev-kurs/helloworld>.

Wenn ihr noch mit keinem dieser Begriffe etwas anfangen könnt, seid beruhight: Wir werden noch so viel mit Git und GitHub arbeiten, dass ihr am Ende dieses Kurses Experten im Umgang damit seid. Befolgt zunächst einfach die Anweisungen in Unterabschnitt 1.2.1 *Workflow mit Git* bis ihr das Beispielprojekt heruntergeladen habt.

¹https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/

2. Öffnet das Xcode-Projekt "**HelloWorld.xcodeproj**" und macht euch mithilfe des Kapitels Xcode im Skript mit der Benutzeroberfläche vertraut. In der Toolbar oben findet ihr auf der linken Seite die Steuerelemente des Compilers. Wählt das *Target HelloWorld* und ein Zielsystem, bspw. den *iPhone 6s Simulator*, und klickt die **Build & Run** Schaltfläche. Das Target wird nun kompiliert und generiert ein *Product*, also unsere App, die im Simulator ausgeführt wird. Das Tastenkürzel für *Build & Run* in Xcode ist **⌘ + R**.
3. Besonders spannend ist diese App natürlich noch nicht. Das ändern wir jetzt spektakulär, indem wir unseren **ersten Swift Code** schreiben um eine Ausgabe hinzuzufügen. Wählt die Datei "**AppDelegate.swift**" links im *Project Navigator* aus.
4. Die Methode `application(_:didFinishLaunchingWithOptions:)` wird zu Beginn der Ausführung der App aufgerufen. Ersetzt den Kommentar dort mit einem Gruß zur Ausgabe in der Konsole:

```

1 func application(application: UIApplication,
2 → didFinishLaunchingWithOptions launchOptions: [NSObject:
3 → AnyObject]?) -> Bool {
4     print("Hello World!")
5     return true
6 }
```

5. Wenn wir unsere App nun erneut mit *Build & Run* **⌘ + R** kompilieren und ausführen, sehen wir den Text "**Hello World!**" in der Konsole. Dazu wird der zweigeteilte Debug-Bereich unten automatisch eingeblendet (s. S. 8, Abb. 2.1). Ist der Konsolenbereich zunächst versteckt, kann er mit der Schaltfläche in der rechten unteren Ecke angezeigt werden. Außerdem wird links automatisch zum Debug Navigator gewechselt, wenn eine App ausgeführt wird, in dem CPU- und Speicherbelastung überwacht werden können und Fehler und Warnungen angezeigt werden, wenn welche auftreten.
6. Wenn ihr ein iOS Gerät dabei habt, verbindet es per USB-Kabel mit eurem Mac und wählt das Gerät in der Toolbar als Zielsystem aus. Mit einem *Build & Run* wird die App nun kompiliert, auf dem Gerät installiert und ausgeführt. In der Konsole erscheint wieder die Ausgabe "**Hello World!**", diesmal direkt vom Gerät ausgegeben.

2.2 Graphisches "Hello World!"

Natürlich wird ein Benutzer unserer App von den Ausgaben in der Konsole nichts mitbekommen. Diese dienen bei der Programmierung hauptsächlich dazu, Abläufe im Code nachzuvollziehen und Fehler zu finden. Unsere App ist also nur sinnvoll, wenn wir die Ausgaben auch auf dem Bildschirm darstellen können.

Relevante Kapitel im Skript: Xcode / Interface Builder

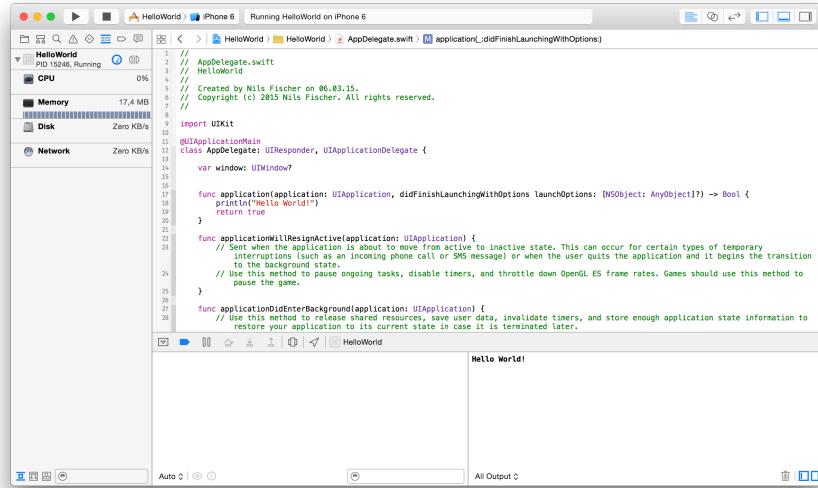


Abbildung 2.1: In der Konsole des Debug-Bereichs werden Ausgaben der laufenden App angezeigt

1. Zur Gestaltung der Benutzeroberfläche oder *User Interface (UI)* verwenden wir ein *Storyboard*. Wählt im Project Navigator die Datei *main.storyboard* aus.
2. Der Editor-Bereich zeigt nun den Interface Builder. In diesem Modus möchten wir häufig eine angepasste Konfiguration des Xcode-Fensters verwenden, es bietet sich also an, mit $\text{⌘} + \text{T}$ einen neuen Tab zu öffnen. Blendet dann mit den Schaltflächen auf der rechten Seite der Toolbar den Navigator- und Debug-Bereich links und unten aus und den Inspektor rechts ein. Wählt dort außerdem zunächst den Standard-Editor, also die linke der drei Schaltflächen (s. S. 9, Abb. 2.2).
3. Unser UI besteht bisher nur aus einer einzigen Ansicht, oder *Scene*. Ein Pfeil kennzeichnet die Scene, die zum Start der App angezeigt wird. Im Inspektor rechts ist unten die Object Library zu finden. Wählt den entsprechenden Tab aus, wenn er noch nicht angezeigt wird (s. S. 9, Abb. 2.2).
4. Durchsucht die Liste von Interfaceelementen nach einem Objekt der Klasse **UILabel**, indem ihr das Suchfeld unten verwendet, und zieht ein Label irgendwo auf die erste Scene. Doppelklickt auf das erstellte Label und tippt "Hello World!".
5. Ein *Build & Run* mit einem iPhone-Zielsystem zeigt diesen Gruß nun statisch auf dem Bildschirm an.
6. Habt ihr das Label im Interface Builder ausgewählt, zeigt der Inspektor Informationen darüber an. Im *Identity Inspector* könnt ihr euch vergewissern, dass das Objekt, was zur Laufzeit erzeugt wird und das Label darstellt, ein Objekt der Klasse **UILabel** ist. Im *Attributes Inspector* stehen viele Optionen zur Auswahl, mit denen Eigenschaften wie Inhalt, Schrift und Farbe des Labels angepasst werden können.
7. Natürlich möchten wir unser UI zur Laufzeit mit Inhalt füllen und den Benutzer mit den Interfaceelementen interagieren lassen können. Zieht ein **UIButton**- und

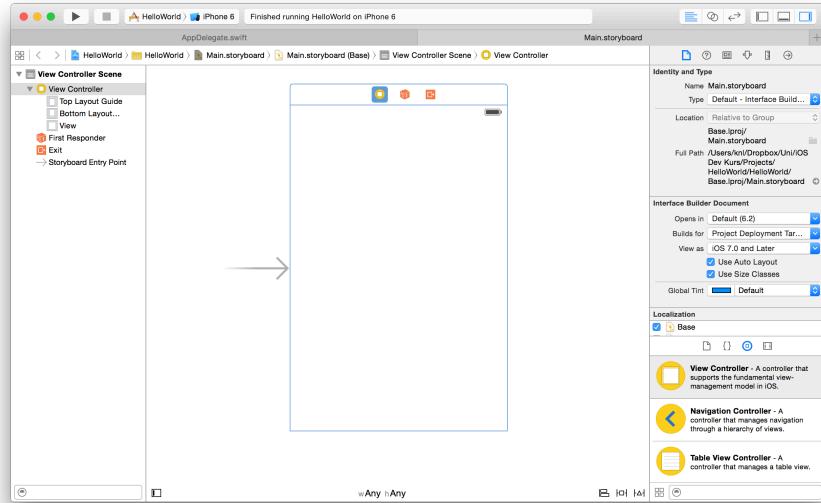


Abbildung 2.2: Für den Interface Builder verwenden wir eine angepasste Fensterkonfiguration mit dem Inspektor anstatt des Navigators

UITextField-Objekt auf die Scene und positioniert sie passend (s. S. 10, Abb. 2.3). Mit dem Attributes Inspector könnt ihr dem Button nun den Titel "**Say Hello!**" geben und für das Text Field einen Placeholder "**Name**" einstellen.

8. Damit sich das Layout an jede Bildschirmgröße automatisch anpasst, verwenden wir nun *Auto Layout*. Die Schaltflächen dazu findet ihr in der unteren rechten Ecke des Interface Builder Editors. Markiert mit gedrückter **[⌘]**-Taste die drei Interfacelemente und klickt auf das Linke der Symbole mit dem Titel *Stack*, sodass die Elemente in eine *Stack View* eingebettet werden. Dieses praktische Objekt positioniert die enthaltenen Elemente automatisch relativ zueinander. Wählt die Stack View aus und konfiguriert im Attributes Inspector *Axis Vertical*, *Alignment Fill*, *Distribution Fill* und *Spacing 8*. Zusätzlich müssen wir Regeln aufstellen, wie die Stack View auf dem Bildschirm positioniert werden soll. Dazu erstellen wir *Constraints* mit den beiden mittleren der Auto Layout Schaltflächen. Befestigt die Stack View links und rechts am Rand und zentriert sie vertikal.
 9. Zur Laufzeit der App wird für jedes im Storyboard konfigurierte Interfaceelement ein Objekt der entsprechenden Klasse erstellt und dessen Attribute gesetzt. Um nun im Code auf die erstellten Objekte zugreifen und auf Benutzereingaben reagieren zu können, verwenden wir *IBOutlets* und *IBActions*.
- Blendet den Inspektor aus und wählt stattdessen den Assistant-Editor (mittlere Schaltfläche) in der Toolbar. Stellt den Modus in der Jump bar auf *Automatic*. Im Assistant wird automatisch die Implementierung des übergeordneten View Controllers eingeblendet (s. S. 11, Abb. 2.4).
10. *View Controller* sind Objekte einer Subklasse von **UIViewController**, die jeweils einen Teil der App steuern. Diese sind zentrale Bestandteile einer App, mit de-

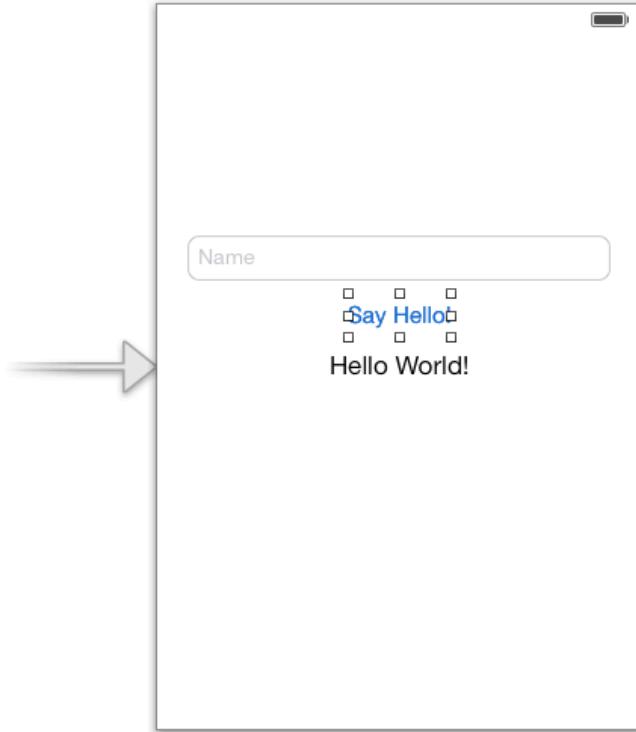


Abbildung 2.3: Mit einem Text Field, einem Button und einem Label erstellen wir ein simples UI

nen wir uns noch detailliert beschäftigen werden. Ein erster View Controller zur Steuerung dieser ersten Ansicht ist im Projekt bereits enthalten.

Fügt dieser Klasse `ViewController`: `UIViewController` Attribute für das `UILabel` und das `UITextField` hinzu und kennzeichnet diese mit `@IBOutlet`. Implementiert außerdem eine mit `IBAction` gekennzeichnete Methode, die aufgerufen werden soll, wenn der Benutzer den `UIButton` betätigt:

```
1 import UIKit
2
3 class ViewController: UIViewController {
4
5     @IBOutlet var nameTextField: UITextField!
6     @IBOutlet var greetingLabel: UILabel!
7
8     @IBAction func greetingButtonPressed(sender: UIButton) {
9         print("Hello World!")
10    }
11
12 }
```

13 @end

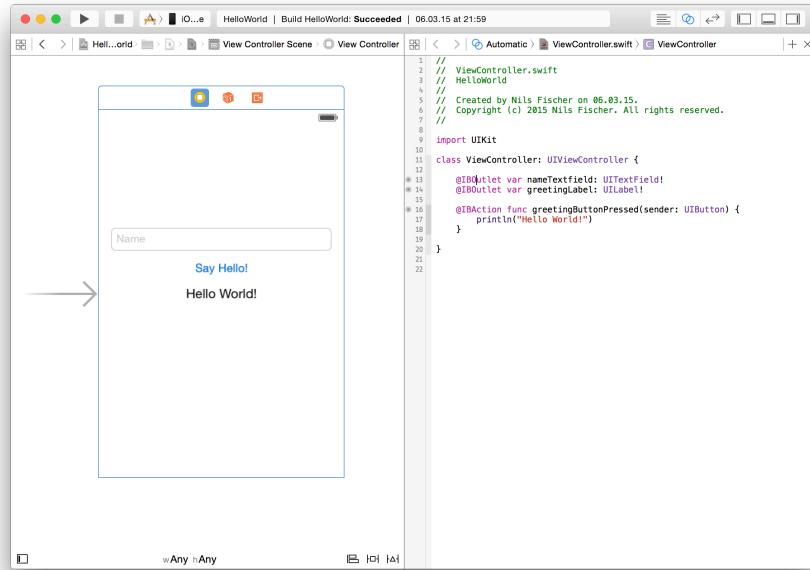


Abbildung 2.4: Mithilfe des Assistants können Interface-Builder und Code nebeneinander angezeigt werden.

11. Nun zieht mit gedrückter **ctrl**-Taste eine Linie von dem Textfeld und dem Label im Interface Builder auf das jeweilige Attribut im Code. Die Codezeile wird dabei blau hinterlegt. Zieht außerdem genauso eine Line von dem Button auf die zuvor definierte Methode. Im Connection Inspector könnt ihr die IBOOutlets und IBActions eines ausgewählten Objekts betrachten und wieder entfernen. Dieser Prozess ist im Skript noch detaillierter beschrieben.
12. Versucht nun einen *Build & Run*. Betätigt ihr den Button, wird die Methode ausgeführt und der Gruß "**Hello World!**" in der Konsole ausgegeben!
13. Um die App nun alltagstauglich zu gestalten, muss dieser Gruß natürlich personalisiert und auf dem Bildschirm angezeigt werden. Dazu verwenden wir das Attribut **text** der Klassen **UITextField** und **UILabel** und zeigen einen personalisierten Gruß an, wenn im Text Field ein Name geschrieben steht:

```

1 @IBAction func greetingButtonPressed(sender: UIButton) {
2     if let name = nameTextField.text where !name.isEmpty {
3         greetingLabel.text = "Hello \(name)!"
4     } else {
5         greetingLabel.text = "Hello World!"
6     }
7 }
```

Nach einem *Build & Run* erhalten wir unser erstes interaktives Interface, in dem ihr im Textfeld einen Namen eintippen könnt und persönlich begrüßt werdet (s. S. 12, Abb. 2.5)!

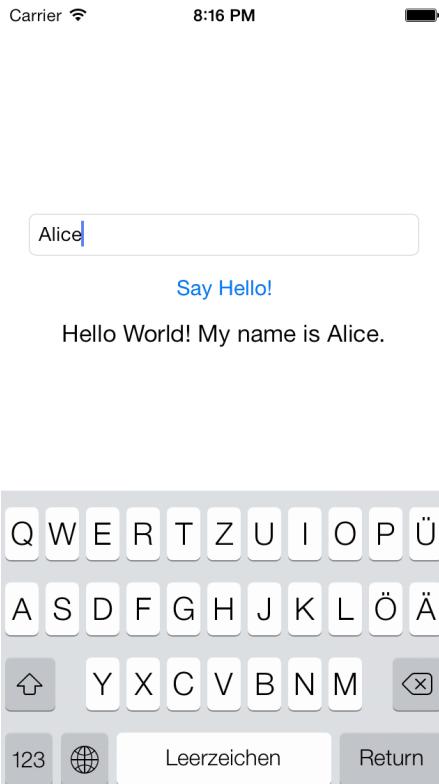


Abbildung 2.5: Drücken wir auf den Button, werden wir persönlich begrüßt. Sehr praktisch!

14. Die App ist fertig! Eure Eltern werden stolz auf euch sein. Gebt mir nun die Gelegenheit, eure Arbeit zu kommentieren. Wir verwendet dazu wieder Git, um die Änderungen, an denen ihr gerade gearbeitet habt, zu speichern, hochzuladen und mir zur Verfügung zu stellen. Befolgt dazu die weiteren Anweisungen in Unterabschnitt 1.2.2, bis ihr mir eine *Pull-Request* geschickt habt.

Übungsaufgaben

1. Simple UI

[2 P.]

Erstellt einen Fork des Repositories <https://github.com/ios-dev-kurs/simpleui> und schreibt eine App mit einigen Interfaceelementen, die etwas sinnvolles tut. Stellt mir das Ergebnis anschließend als Pull-Request zur Verfügung.

Implementiert eines der folgenden Beispiele oder eine eigene Idee. Ich freue mich auf kreative Apps!

Counter Auf dem Bildschirm ist ein Label zu sehen, das den Wert eines Attributs `var count: Int` anzeigt, wenn eine Methode `updateLabel` aufgerufen wird. Buttons mit den Titeln "+1", "-1" und "Reset" ändern den Wert dieses Attributs entsprechend und rufen die `updateLabel`-Methode auf.

BMI Nach Eingabe von Gewicht m und Größe l wird der Body-Mass-Index^[2] $BMI = m/l^2$ berechnet und angezeigt.

RGB In drei Textfelder kann jeweils ein Wert zwischen 0 und 255 für die Rot-, Grün- oder Blau-Komponenten eingegeben werden. Ein Button setzt die Hintergrundfarbe `self.view.backgroundColor` entsprechend und ein weiterer Button generiert eine zufällige Hintergrundfarbe. Ihr könnt noch einen `UISwitch` hinzufügen, der einen Timer ein- und ausschaltet und damit die Hintergrundfarbe bei jedem Timerintervall zufällig wechselt (s. Hinweis).

Hinweise:

- In der nächsten Vorlesung lernen wir die Objektorientierte Programmierung in Swift systematisch. Orientiert euch für diese Aufgabe an der *HelloWorld* App und versucht die Funktionalität mit den folgenden Hinweisen zu implementieren. Wenn ihr nicht weiter kommt, schickt mir eine Pull-Request mit einem Kommentar und ich helfe euch.
- Das Attribut `text` von `UILabel` und `UITextField` gibt eine *optionale* Zeichenkette `String?` zurück. Ihr werdet euch mit solchen *Optionals* solange herumärgern, bis ihr sie zu schätzen lernt. Verwendet die *Optional Binding Syntax* um das Optional zu entpacken:

```

1 if let name = nameTextfield.text {
2     // name existiert und kann verwendet werden
3 } else {
4     // nameTextfield.text hat keinen Wert
5 }
```

- Einen `String` könnt ihr schnell in eine ganze Zahl `Int` oder eine Dezimalzahl `Double` umwandeln. Da dies fehlschlagen kann, gibt auch diese Operation einen Optional `Int?` bzw. `Double?` zurück, den wir entpacken müssen:

^[2]<http://de.wikipedia.org/wiki/Body-Mass-Index>

```

1 // Sei text ein String
2 if let number = Double(text) {
3     // text konnte in eine Zahl number umgewandelt werden
4 }
```

- Definiert ein Attribut wie `var count: Int` mit einem Startwert:

```

1 class ViewController: UIViewController {
2
3     var count: Int = 0
4
5     // ...
6 }
```

- Natürlich gibt es die grundlegenden Rechenoperationen `+ - * /` in Swift. Diese Operationen können mit der Zuweisung zu einer Variablen verbunden werden, um bspw. eine Variable `count` um `1` zu erhöhen:

```
1 count += 1
```

- Eine Farbe wird durch die Klasse `UIColor` repräsentiert. Der *Initializer* `UIColor(red:green:blue:alpha:)` akzeptiert jeweils Werte zwischen 0 und 1:

```
1 let color = UIColor(red: 1, green: 0, blue: 0, alpha: 1) // rot
```

- Die Funktion `arc4random_uniform(n)` gibt eine Pseudozufallszahl x mit $0 \leq x < n$ aus.
- Wenn ein `UISwitch` betätigt wird, kann das Event genauso mit einer IBAction verbunden werden wie das eines `UIButton`. Mit einem Attribut `var randomTimer: NSTimer?` können wir dann die Methode für das zufällige Wechseln der Hintergrundfarbe implementieren:

```

1 var randomTimer: NSTimer?
2
3 @IBAction func switchValueChanged(sender: UISwitch) {
4     if sender.on {
5         randomTimer =
6             NSTimer.scheduledTimerWithTimeInterval(0.15, target:
7                 self, selector: "randomButtonPressed:", userInfo:
8                 nil, repeats: true)
9     } else {
10        randomTimer?.invalidate()
```

```
8         randomTimer = nil
9     }
10 }
```

Somit wird periodisch die Methode `randomButtonPressed(_:)` aufgerufen, die natürlich implementiert sein muss.

Kapitel 3

A Swift Tour

Für unsere ersten Apps hat eine gute Portion Intuition für ein wenig Swift Code ausgereicht. Bevor wir tiefer in die App-Programmierung einsteigen beschäftigen wir uns einmal genauer mit der Programmierung in Swift.

Apple bietet mit dem Buch *The Swift Programming Language* eine hervorragende Dokumentation und ein einführendes Kapitel mit den Namen *A Swift Tour*. Das Buch findet ihr sowohl online [¹] als auch im iBooks Store [²] immer in aktueller Version. Auch das letzte Kapitel *Language Reference* ist sehr spannend wenn ihr euch für den detaillierten Aufbau der Sprache und deren Grammatik interessiert.

Löst die Übungsaufgaben mit eurem Wissen aus der Vorlesung und den zugehörigen Materialien auf der Vorlesung Webseite. Zieht zuerst das Buch *The Swift Programming Language* zu Rate wenn ihr nicht weiterkommt. Schickt mir bei weiteren Fragen eine Pull-Request oder einen Xcode Playground per Email.

Xcode Playgrounds eignen sich hervorragend um Swift Code auszuprobieren und um Code zu schreiben der die Infrastruktur einer App nicht erfordert, wie bspw. die Übungsaufgaben *Fibonacci*, *Primzahlen* und *Poker*. Erstellt einen Playground mit `File > New > Playground...` oder `⌘ + ⌘ + ⌘ + N`.

Die Übungsaufgaben *Fibonacci* und *Primzahlen* sind optional und an Kursteilnehmer gerichtet, die noch wenig oder keine Programmierkenntnisse mitbringen. Auch erfahrene Programmierer können aber anhand dieser Aufgaben die Swift Syntax kennenlernen und versuchen die Aufgaben so *swifty* wie möglich zu lösen.

¹https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/

²<https://itunes.apple.com/de/book/swift-programming-language/id881256329?mt=11>

Übungsaufgaben

2. Optional: Fibonacci

[+1 P.]

Schreibt in einem Xcode Playground einen Algorithmus der alle Folgenglieder $F_n < 1000$ der Fibonaccifolge

$$F_n = F_{n-1} + F_{n-2} \quad (3.1)$$

$$F_1 = 1, F_2 = 2 \quad (3.2)$$

in der Konsole ausgibt.

Hinweis: Versucht's mit einer `while`-Schleife und zwei Variablen für die letzten beiden Folgenglieder, die außerhalb der Schleife definiert wurden. Wer die Aufgabe richtig *swifty* lösen will kann stattdessen ein `struct`: `FibonacciSequence` schreiben welches das `SequenceType` Protokoll erfüllt.

3. Optional: Primzahlen

[+1 P.]

- a) Schreibt eine Funktion `func isPrimeNumber(n: Int) -> Bool` die eine Zahl annimmt und `true` zurückgibt, wenn diese eine Primzahl ist, andernfalls `false`.

Hinweis: Iteriert in einer `for`-Schleife durch alle Zahlen von 2 bis n: `for i in 2... Überspringt den Schleifenschritt mit continue wenn i gleich n ist. Prüft sonst mit dem Modulo-Operator % den Rest der Division $n \% i$ der beiden Zahlen. Ist dieser 0 so ist n durch i teilbar und ihr könnt false zurückgeben: return false. Gebt sonst nach dem Durchlauf der Schleife true zurück.`

- b) Schreibt dann eine Funktion `func primeNumbersUpTo(maxNumber: Int) -> [Int]` die alle Primzahlen bis maxNumber als Liste `[Int]` (kurz für `Array<Int>`) zurückgibt.

Hinweis: Erstellt zuerst eine leere Liste von `Ints`: `var primeNumbers: [Int] = []`. Iteriert dann in einer `for`-Schleife durch die Zahlen `1...maxNumber` und fügt die Zahl der Liste mit `primeNumbers.append(n)` hinzu wenn sie `isPrimeNumber(n)` erfüllt. Gebt die Liste mit `return primeNumbers` nach Schleifendurchlauf zurück. Richtig *swifty* könnt ihr die Aufgabe auch in einer Zeile lösen indem ihr die Methode `filter` von `SequenceType` verwendet.

4. Chatter

[2 P.]

In dieser Aufgabe schreiben wir zusammen an einer App!

Forkt das Repository <https://github.com/ios-dev-kurs/chatter> und erstellt eine Pull Request um eure Lösung einzureichen oder Fragen zu stellen.

- i. Die *Chatter* App ist in der "`README.md`" Datei beschrieben. Ihr könnt euch die Projektdateien anschauen und die App im Simulator oder auf euren Geräten ausführen und ausprobieren. Wenn euch interessiert, wie die App aufgebaut ist, lest die "`README.md`" und die Kommentare im Code.

- ii. Ihr habt nun sicherlich erkannt worum es in der App geht: Instanzen verschiedener Subklassen von Chatter chatten miteinander. Dabei überschreiben die Subklassen jeweils nur die Implementierung weniger Methoden, die in der Chatter Klasse dokumentiert sind.

Eure Aufgabe ist es nun, eine eigene Subklasse zu schreiben und damit euren Beitrag zu dieser App zu leisten! Ihr könnt einen bekannten Charakter darstellen oder einen Neuen erschaffen. Ich übernehme eure Pull-Request dann in das Original-Repository, sodass euer Charakter mit denen aller anderen Kursteilnehmer chatten kann.

Erstellt dazu mit eine neue ".swift"-Datei mit dem Namen eures Charakters und platziert sie im Xcode Project Navigator unter . Orientiert euch an Yoda.swift um eure neue Subklasse von Chatter zu implementieren.

- iii. Überschreibe die relevanten Methoden in eurer Subklasse wie in der "**README.md**" Datei beschrieben. Hier könnt ihr einfach zufällige Chatnachrichten generieren, oder auch komplexere Mechaniken einbauen, sodass eine etwas natürlichere Konversation zustande kommt.

In eurer eigenen Subklasse könnt ihr dabei beliebig Code schreiben und bspw. Attribute einführen, um den Zustand eures Charakters darzustellen, wenn ihr möchtet. Er oder sie (oder es?) könnte bspw. mit jeder Nachricht wütender werden oder dergleichen.

- iv. Sichert eure Änderungen regelmäßig in Commits, wenn der Code fehlerfrei kompiliert. Achtet darauf nur Änderungen eurer Subklasse und nur wenn nötig Änderungen in anderen Dateien zu committen. Die "**project.pbxproj**" Datei enthält Informationen zu den Projektdateien - da ihr neue Dateien hinzugefügt habt, gehört diese zum Commit dazu.
- v. Mit eurem Fork des Repositories auf GitHub könnt ihr eure Änderungen jederzeit abgleichen. Bei der Gelegenheit bietet es sich an auch die neuesten Änderungen aus dem Original-Repository herunterzuladen, sodass ihr die neuen Charaktere der anderen Kursteilnehmer erhaltet:

```
1 git pull https://github.com/ios-dev-kurs/chatter.git master
```

Die pull Operation versucht, die heruntergeladenen Änderungen mit den lokalen Änderungen zusammenzuführen. Das klappt nicht immer ohne Konflikte. Da jeder von euch dem Projekt eine Datei hinzufügt ändert sich jeweils die "**project.pbxproj**" Datei. Treten Konflikte auf, müsst die die Datei in einem Texteditor öffnen und nach den Konfliktmarkierungen suchen:

```
1 <<<<< HEAD:  
2 # lokaler Code vor dem Merge  
3 =====
```

```
4 # durch den Merge veränderter Code
5 >>>>>
```

Behebt den Konflikt indem ihr die Konfliktmarkierungen löscht und den Code dazwischen gegebenenfalls anpasst. Dann könnt ihr den Merge committen:

```
1 git add --all
2 git commit
```

Euer Repository enthält dann sowohl den aktuellen Stand des Original-Repositories, als auch eure Änderungen.

- vi. Wenn ihr mit eurer neuen `Chatter` Subklasse zufrieden seid schickt mir eine Pull-Request. So werden eure Änderungen in das Original-Repository integriert und tauchen auch bei den anderen Teilnehmern auf, wenn diese das nächste mal einen `git pull` durchführen.

Ich bin gespannt auf eure Implementierungen!

5. Poker

[3 P.]

In dieser Aufgabe berechnen wir die Wahrscheinlichkeit für einen *Flush* beim Poker.

Forkt das Repository <https://github.com/ios-dev-kurs/poker> und erstellt eine Pull Request um eure Lösung einzureichen oder Fragen zu stellen.

- i. Zuerst modellieren wir die Spielkarten. Eine Karte `Card` hat immer eine *Farbe Suit* (Karo, Herz, Pik oder Kreuz) und einen *Rang Rank* (2 bis 10, Bube, Dame, König oder Ass).

Wir modellieren `Card` als Struct, und `Suit` und `Rank` als Enums. Warum verwenden wir keine Klasse für `Card`? Warum eignet sich ein Enum so hervorragend für `Suit` und `Rank`? Beantwortet diese Fragen kurz stichwortartig in einem Kommentar im Playground.

- ii. Schreibt zwei Enums `enum Suit: Int` und `enum Rank: Int` mit ihren jeweiligen Fällen (`case diamonds` usw.). Bei den Rängen 2 bis 10 schreibt ihr am besten die Zahl aus (`case two` usw.).

Implementiert jeweils eine *Computed Property* `var description: String` in der ihr mithilfe einer `switch`-Abfrage für jeden Fall ein Symbol zurückgibt. **Tipp:** Für Karo, Herz, Pik und Kreuz gibt es Unicode-Symbole^[3]! Außerdem verlangt das Protokoll `CustomStringConvertible` nur das Attribut `description`, schreibt also z.B. `enum Suit: Int, CustomStringConvertible` damit das Symbol in `print`s verwendet wird.

³http://en.wikipedia.org/wiki/Playing_cards_in_Unicode

Schreibt dann einen `struct Card` mit zwei Attributen `let suit: Suit` und `let rank: Rank`, sowie einer *Computed Property* `var description: String`, die einen aus Farbe und Rang zusammengesetzten String zurückgibt. Lasst auch `Card` das `CustomStringConvertible` Protokoll erfüllen.

- iii. Nun können wir eine Poker Hand modellieren. Schreibt den `struct PokerHand` mit einem Attribut `let cards: [Card]` und einer *Computed Property* `var description: String`, die die `description` der Karten kombiniert.

Um einfach zufällige Poker Hände generieren zu können, implementiert einen Initializer `init()`, der eine Hand aus fünf zufälligen Karten erstellt. **Wichtig:** Da aus einem Deck von paarweise verschiedenen Karten gezogen wird, darf keine Karte doppelt vorkommen.

Hinweise:

- Da wir `Suit` und `Rank` von `Int` abgeleitet haben, können wir Zufallszahlen generieren und die Enums daraus erstellen:

```

1 let rndSuit = Suit(rawValue: Int(arc4random_uniform(4)))!
2 let rndRank = Rank(rawValue: Int(arc4random_uniform(13)))!
3 let rndCard = Card(suit: rndSuit, rank: rndRank) // Eine
   → zufällige Spielkarte

```

- Die Funktion `contains` könnte hilfreich sein, um das Vorhandensein von Karten zu überprüfen. Um diese mit `Card` verwenden zu können muss `Card` das `Equatable` Protokoll erfüllen. Schreibt `extension Card: Equatable {}` und dann außerhalb:

```

1 func ==(lhs: Card, rhs: Card) -> Bool {
2     return lhs.suit == rhs.suit && lhs.rank == rhs.rank
3 }

```

Erstellt ein paar Poker Hände und lasst euch die `description` ausgeben. Habt ihr etwas gutes gezogen?

- iv. Implementiert nun ein weiteres Enum `enum Ranking: Int` mit den Fällen `case highCard, flush, straightFlush` usw., die ihr bspw. auf Wikipedia^[4] findet.

Fügt dann dem `struct PokerHand` eine Computed Property `var ranking: Ranking` hinzu. Implementiert hier einen Algorithmus, der prüft, ob ein *Flush* vorliegt. Dann soll `.flush` zurückgegeben werden, ansonsten einfach `.highCard`.

⁴http://en.wikipedia.org/wiki/List_of_poker_hands

- v. Wir können nun einige tausend Hände generieren und die Wahrscheinlichkeit für einen Flush abschätzen. Fügt einfach folgenden Code am Ende des Playgrounds ein:

```

1  var rankingCounts = [Ranking : Int]()
2  let samples = 1000
3  for i in 0...samples {
4      let ranking = PokerHand().ranking
5      if rankingCounts[ranking] == nil {
6          rankingCounts[ranking] = 1
7      } else {
8          rankingCounts[ranking]! += 1
9      }
10 }
11
12 for (ranking, count) in rankingCounts {
13     print("The probability of being dealt a
14         \((ranking.description) is \(Double(count) /
15             Double(samples) * 100)%")
16 }
```

Die Ausführung kann etwas dauern, justiert ggfs. `samples`. Stimmt die Wahrscheinlichkeit ungefähr mit der Angabe auf Wikipedia überein?

- vi. **Extra:** Ihr könnt das Programm nun noch erweitern und versuchen, die anderen Ränge zu überprüfen. Dabei könnten Hilfsattribute wie `var hasFlush: Bool` oder `var pairCount: Int` nützlich sein. Bekommt es jemand es jemand hin, eine Funktion zu schreiben, die zwei Hände vergleicht und den Sieger bestimmt? **Tipp:** Dazu könnte es hilfreich sein, die Fälle des `enum: Ranking` um *Associated Attributes* zu erweitern.

Kapitel 4

iOS App Architektur

Wir haben nun die Grundlagen der Programmierung in Swift gelernt. Jetzt können wir uns der Programmierung komplexerer Apps zuwenden. Dabei lernen wir die *Architektur* von iOS Apps kennen und verwenden Konzepte zur Strukturierung unseres Programmcodes, die für größerere Softwareprojekte notwendig sind.

Die grundlegende Architektur einer iOS App illustriert das Repository <https://github.com/ios-dev-kurs/bare> und der Abschnitt *iOS App Lifecycle* im Skript. Schaut euch insbesondere auch die Commitfolge `git log` des Repositories an. Jeder Commit stellt einen Schritt vom minimal ausführbaren Code bis zu einer funktionsfähigen App mit Storyboard dar. Ein Commit wird durch seinen *Hash* identifiziert, der auch von `git log` ausgegeben wird. Diesen könnt ihr verwenden, um das Verzeichnis in den Zustand zu einem bestimmten Commit zu versetzen:

```
1 git checkout e35cec8e71ffb87d19b837cf48c12837329a6d82 # z.B. Hash des  
→ ersten Commits
```

Sobald unsere App startet und ein Storyboard lädt sind liegt die Verantwortung bei uns. Unsere Aufgabe ist nun das *Software Engineering*. Viele Konzepte sind von der Plattform und Programmiersprache unabhängig gültig (siehe bspw. das *DRY - Don't repeat yourself* Prinzip [1]). Ziel ist, unseren Programmcode übersichtlich, flexibel und erweiterbar zu strukturieren. Damit vermeiden wir automatisch Fehler und verlieren uns nicht in der Komplexität des Codes.

Die iOS App Entwicklung orientiert sich konsequent am *Model-View-Controller Konzept* der Programmierung oder Varianten dieses Konzepts. Es ist nicht nur in Apples Frameworks wie `UIKit` rigoros umgesetzt sondern stellt auch die Grundlage für die weitere Konzeption unserer Apps dar und wird auch in vielen anderen Bereichen der Softwareentwicklung verwendet. Das Konzept ist im Skript beschrieben und sollte bei Entscheidungen zur Architektur einer App stets zu Rate gezogen werden.

Relevante Kapitel im Skript: Das Model-View-Controller Konzept

¹https://de.wikipedia.org/wiki/Don%27t_repeat_yourself

Übungsaufgaben

6. Lifetime

[2 P.]

In dieser Aufgabe zeigen wir für die Kontakte auf unseren iOS Geräten die Zeit seit ihrem Geburtstag an.

Forkt das Repository <https://github.com/ios-dev-kurs/lifetime> und erstellt eine Pull-Request um eure Lösung einzureichen oder Fragen zu stellen.



Model



Controller

- a) Die *Model*-Komponente der App ist bereits implementiert. Wir verwenden das Contacts Framework von Apple, das Zugriff auf die Kontakte ermöglicht. In "[Lifetime.swift](#)" erweitern wir die Klasse CNContact um ein Computed Attribute `lifetime: NSTimeInterval?`.

Um auf Kontakte zuzugreifen stellen wir Anfragen an einen `CNContactStore`. Das AppDelegate erstellt einen solchen und reicht ihn an den ContactListViewController weiter. Dieser lädt die Kontakte und soll sie nun anzeigen. Dazu verwaltet er eine `UITableView`, die nach dem *Delegate*-Prinzip nach Bedarf Anfragen nach Zahl und Inhalt der anzuseigenden Zeilen stellt. Diese Anfragen muss der ContactListViewController beantworten. Eure erste Aufgabe ist, zu diesem Zweck das `UITableViewDatasource` Protokoll zu implementieren.

Implementiert das Protokoll in einer Erweiterung in "[ContactListViewController.swift](#)". Die folgenden drei Methoden muss jede Implementierung des Protokolls mindestens bereitstellen:

```
1 extension ContactListViewController {
2
3     override func numberOfSectionsInTableView(tableView:
4         UITableView) -> Int {
5         return 1 // Wir zeigen die Kontakte zunächst in einer
6         // einzelnen Section
7     }
8
9     override func tableView(tableView: UITableView,
10        numberOfRowsInSection section: Int) -> Int {
11         return contacts.count // Für jeden Kontakt soll eine
12         // Zeile angezeigt werden
13     }
14
15     override func tableView(tableView: UITableView,
16        cellForRowAtIndexPath indexPath: NSIndexPath) ->
17        UITableViewCell {
18         // VIEW-Komponente: Frage die Table View nach einer
19         // wiederverwendbaren Zelle
20     }
21 }
```

```

13         let cell = table-
14             ↵ View.dequeueReusableCellWithIdentifier("LifetimeCell",
15             ↵ forIndexPath: indexPath) as! LifetimeCell
16 // MODEL-Komponente: Bestimme den Kontakt für diese
17             ↵ Zeile
18         let contact = contacts[indexPath.row]
19 // CONTROLLER-Komponente: Konfiguriere die Zelle nach
20             ↵ dem Kontakt
21         cell.configureForContact(contact)
22         if contact.lifetime != nil {
23             cell.selectionStyle = .Default
24             ↵ cell.accessoryType = .DisclosureIndicator
25         } else {
26             ↵ cell.selectionStyle = .None
27             ↵ cell.accessoryType = .None
28         }
29     }
30 }

```



View

- b) Es fehlt noch die *View*-Komponente, also die Zelle die wir oben bereits verwenden. Erstellt eine neue Datei "LifetimeCell.swift" und platziert sie im Project Navigator unter *View*. Implementiert darin die Klasse `class LifetimeCell: UITableViewCell`:

```

1 import UIKit
2 import Contacts
3
4 class LifetimeCell: UITableViewCell {
5
6     func configureForContact(contact: CNContact) {
7        .textLabel?.text =
8             ↵ CNContactFormatter.stringFromContact(contact, style:
9                 ↵ .FullName)
10        if let lifetime = contact.lifetime {
11            let lifetimeFormatter = NSDateComponentsFormatter()
12            lifetimeFormatter.allowedUnits = .Day
13            lifetimeFormatter.unitsStyle = .Full
14            detailTextLabel?.text = lifetimeFormatter.
15                 ↵ stringFromTimeInterval(lifetime)
16        } else {
17            detailTextLabel?.text = nil
18        }
19    }
20 }

```

18 }

Wir gestalten die Zelle im "Main.storyboard". Zieht aus der Object Library eine `UITableViewCell` auf die `UITableView`. Ändert im Attributes Inspector ihren *Style* zu *Right Detail* und gebt ihr den *Identifier* "`LifetimeCell`". Wählt dann den Identity Inspector und ändert die Klasse der Zelle zur gerade implementierten `LifetimeCell`. Beide Schritte sind notwendig, sodass die zuvor implementierte `tableView(_:cellForRowAtIndexPath:)` Methode fehlerfrei ausgeführt wird.

- c) Nun könnt ihr die App ausführen und seht bereits eine Liste der Kontakte!



Controller

Wenn wir einen Kontakt antippen soll dessen Detailansicht angezeigt werden, die von einem anderen View Controller verwaltet wird. Im Skript könnt ihr euch über die *View Controller Hierarchy* informieren. Der `ContactListViewController` wird bereits von einem `UINavigationController` verwaltet, den wir nun verwenden um einen `ContactDetailViewController` anzuzeigen.

Im "Main.storyboard" können wir den Übergang zwischen den View Controllern durch eine *Storyboard Segue* implementieren. Zieht mit gedrückter `ctrl`-Taste eine Line von der "`LifetimeCell`" zum `ContactDetailViewController` und wählt im Popup *Show*. Wählt die so erstellte Segue aus und gebt ihr im Attributes Inspector den *Identifier* "`showContactDetail`".

- d) Schließlich müssen wir den ausgewählten Kontakt im `ContactListViewController` noch an den `ContactDetailViewController` weitergeben. Dazu dient die `prepareForSegue(_:sender:)` Methode:

```

1 override func prepareForSegue(segue: UIStoryboardSegue, sender:
2     AnyObject?) {
3     switch segue.identifier! {
4
5         case "showContactDetail":
6             guard let indexPath =
7                 self.tableView.indexPathForSelectedRow else { break }
8             let contact = contacts[indexPath.row]
9             let contactDetailViewController =
10                segue.destinationViewController as!
11                ContactDetailViewController
12             contactDetailViewController.contact = contact
13
14         default:
15             break
16     }
17 }
```

Sehr ähnlich könnt ihr noch die `shouldPerformSegueWithIdentifier(_:sender:)` Methode implementieren und für "`"showContactDetail"` `contact.lifetime != nil` zurückgeben, sodass die Detailansicht nur für Kontakte mit gültigem Geburtstag angezeigt wird.

- e) Unsere App zeigt uns jetzt, wie lang unsere Kontakte schon leben!

Statt den *Right Detail* Stil für die `LifetimeCell` zu verwenden könnt ihr die Zelle natürlich auch selbst gestalten. Wenn ihr noch Erweiterungen einbauen möchtet für die ihr weitere Attribute von `CNContact` benötigt, wie bspw. `CNContactImageDataKey`, müsst ihr diese der Liste `requiredContactKeysToFetch` in `ContactListViewController` hinzufügen, sodass sie ebenfalls aus dem `CNContactStore` geladen werden.

Schickt eine Pull-Request wenn ihr mit eurer App zufrieden seid oder um Fragen zu stellen.

7. Seasonizer

[3 P.]

Mit den Apps der vergangenen Vorlesungen haben wir die Grundlagen der Programmierung für iOS Geräte und einige wichtige Architekturkonzepte kennengelernt. Mit diesem Wissen lässt sich bereits ein Großteil der zur Verfügung stehenden Frameworks anwenden und viele Funktionen, die iOS Geräte bieten, in unsere eigenen Apps integrieren.

Zusammenfassend implementieren wir in dieser App noch einmal die grundlegende iOS App Architektur. Dabei lernen wir zusätzlich die Integration von Multi-Touch Gesten und der Kamera kennen, um beim nächsten Familienurlaub nicht nur mit unseren neu erworbenen Programmierfähigkeiten anzugeben, sondern auch noch Fotos von Familienmitgliedern sommerlich dekorieren zu können (s. S. 27, Abb. 4.1).

Features der Seasonizer App:

- Die App besitzt eine Hauptansicht mit Navigation Bar und Toolbar. Den Inhalt füllt eine Image View für das Foto und eine darüberliegende View mit transparentem Hintergrund für die Accessories.
- Es gibt einen Button in der Toolbar mit dem ein Foto von der Kamera oder Foto Bibliothek ausgewählt werden kann, das anschließend von der Image View angezeigt wird.
- Ein weiterer Button zeigt modal eine Liste von Accessories an. Wird ein Accessory ausgewählt, wird es der Hauptansicht hinzugefügt.
- Die Accessories lassen sich mit Gesten verschieben, skalieren, drehen und löschen.
- Mit einem Action Button kann das Bild über verschiedene Kanäle wie Nachrichten, Email oder Facebook verteilt werden.
- Die Elemente wie Bild und Accessories werden gespeichert und beim nächsten Start der App wiederhergestellt.



Abbildung 4.1: Mit der Seasonizer App lassen sich Freunde und Familienmitglieder sommerlich dekorieren.

- Ein Toolbar Button dient dem Zurücksetzen der Benutzeroberfläche.

Forkt das Repository <https://github.com/ios-dev-kurs/seasonizer> und vervollständigt die App. Erstellt eine Pull-Request um eure Lösung einzureichen oder Fragen zu stellen.

Hinweise

- **Navigation Bar und Toolbar** lassen sich mit einem Navigation Controller sehr einfach anzeigen. Dieser besitzt ein Attribut `Shows Toolbar`, das im Interface Builder aktiviert werden kann. Wenn wir dann Buttons als Objekte der `UIBarButtonItem` Klasse einem View Controller hinzufügen zeigt der Navigation Controller diese in der Toolbar an. `UIBarButtonItem` bietet bereits viele Stile wie `UIBarButtonSystemItemCamera` zur Auswahl im Attributes Inspector an. Für's Layout gibt es außerdem den Stil `Flexible Space`.
- Um den `AccessoryListViewController` modal mit Titelleiste anzuzeigen verpacken wir ihn wiederum in einem Navigation Controller (s. S. 28, Abb. 4.2). Gebt die Liste der Accessories in der `prepareForSegue(_:sender:)` Methode an den `AccessoryListViewController` weiter. Implementiert darin dann das `UITableViewDatasource` Protokoll.

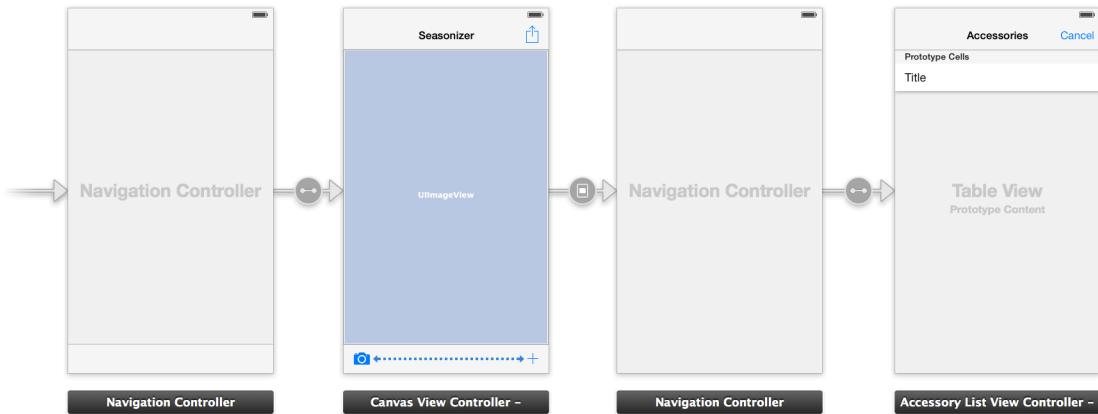


Abbildung 4.2: Navigation Controller bieten eine einfache Möglichkeit, Navigationsleisten und Toolbars anzuzeigen

- Verwendet *Unwind Segues* um den **AccessoryListViewController** wieder zu verlassen. Definiert dazu eine `@IBAction func unwindToCanvas(segue: UIStoryboardSegue)` Methode im **CanvasViewController**. Erstellt die Unwind Segues dann von einem *Cancel* Button und von der **AccessoryCell** in dem ihr eine Verbindung zur *Exit* Schaltfläche zieht. Gebt den Segues jeweils einen Identifier. In der `prepareForSegue(_:sender:)` Methode des **AccessoryListViewController** könnt ihr dann das `selectedAccessory` setzen und es in der `unwindToCanvas(_:)` Methode wieder auslesen.

Kapitel 5

Methoden

Wir können nun durch sorgfältige Konstruktion der View und View Controller Hierarchien bereits recht komplexe iOS Apps aufbauen. Das Model View Controller Konzept hilft uns dabei den Programmcode zu strukturieren. Jetzt können wir Lösungen zu verbreiteten Aufgaben der Softwareentwicklung kennen lernen und uns mit Methoden des Software Engineering befassen.

5.1 Kommunikation mit einer REST API

Für eine Software, die eine Aufgabe erfüllt, sind deren *Schnittstellen* häufig noch wichtiger als die Funktionsweise des Programms. Durch Schnittstellen können andere Entwickler Teilprobleme an spezialisierte und getestete Software auslagern, ohne deren Implementierungsdetails kennen zu müssen. Nur so können wir Apps schreiben, die eine bestimmte Aufgabe hervorragend erfüllen und das Ergebnis dann weitergeben.

Eine iOS App kann Schnittstellen zu anderen Apps oder Services des Betriebssystems anbieten. Bspw. kann sich ein Social Network darauf spezialisieren, dem Benutzer einen Ort vorzuschlagen, und diesen dann an eine Navigationsystem-App weitergeben, die auf das Routing spezialisiert ist. Solche Schnittstellen werden unter iOS als *Deep Links* mit *URL Schemata* realisiert.

Häufig nutzen iOS Apps Schnittstellen. Sie bieten eine Benutzeroberfläche an, um eine Schnittstelle zu konsumieren. Dabei handelt es sich meist um einen Server, der plattformunabhängig Daten in Form einer REST API bereitstellt. Der Server akzeptiert HTTP Anfragen in Form einer URL, und verarbeitet diese oder gibt Daten in einem Format wie JSON zurück. URL und JSON folgen dabei einem definierten Schema.



Übungsaufgaben

8. API Client

[2+3 P.]

Mit der *API Client* App implementieren wir eine Benutzeroberfläche für eine REST API. Wir konzentrieren uns dabei auf die *Model* und *Controller* Komponenten unserer App. Neben asynchronen Netzwerkanfragen lernen wir auch den *Error Handling* Mechanismus und *Generics* in Swift kennen. Außerdem integrieren wir Frameworks anderer Entwickler in unsere App. Indem wir die Schnittstellen dieser Frameworks nutzen können wir auf sorgfältig konzipierten, funktionsfähigen und ausgiebig getesteten Code zurückgreifen. So müssen wir uns nicht erneut mit einem Problem befassen, für das es bereits hervorragende Lösungen gibt, und können uns stattdessen auf die Funktionalität unseres Programms konzentrieren.

Im *APIClient* Repository [1] findet ihr ein Projekt, in das bereits vier hervorragende Open-Source Frameworks mithilfe des Dependency Managers *CocoaPods* [2] integriert sind:

Moya^[3] Network abstraction layer written in Swift.

Freddy^[4] A reusable framework for parsing JSON in Swift.

Alamofire^[5] Elegant HTTP Networking in Swift

AwesomeCache^[6] Delightful on-disk cache (written in Swift)

CocoaPods lädt die in der "**Podfile**" Datei angegebenen Frameworks in das "**Pods/**" Verzeichnis und integriert sie zusammen mit dem Projekt in einen Xcode Workspace. Verwendet also immer den "**APIClient.xcworkspace**" statt des "**.xcodeproj**".

Euch steht es frei, eine Benutzeroberfläche für eine API eurer Wahl zu schreiben. Eine Sammlung von vielen interessanten APIs findet ihr bspw. hier [7]:

- Natürlich viele **Wetter** APIs [8]
- Die APIs der **NASA** [9]
- Spieleentwickler wie **Blizzard** [10]
- Comic-Helden von **Marvel** [11]
- Computational Services wie **Wolfram Alpha** [12]
- Nährstoffdaten für Lebensmittel der **National Nutrient Database** [13]

¹<https://github.com/knly/apiclient/>

²<https://cocoapods.org>

⁷https://www.reddit.com/r/webdev/comments/3wrszc/what_are_some_fun_apis_to_play_with/

⁸<https://developer.forecast.io>

⁹<https://data.nasa.gov/developer>

¹⁰<https://dev.battle.net/io-docs>

¹¹<http://developer.marvel.com/docs>

¹²<http://products.wolframalpha.com/api/>

¹³<https://ndb.nal.usda.gov/ndb/api/doc>

- Vom gleichen Entwickler wie die **PokeAPI** [14] aus der Vorlesung (s. S. 31, Abb. 5.1) ist die Star Wars API **SWAPI** [15] und eignet sich ebenfalls sehr gut.

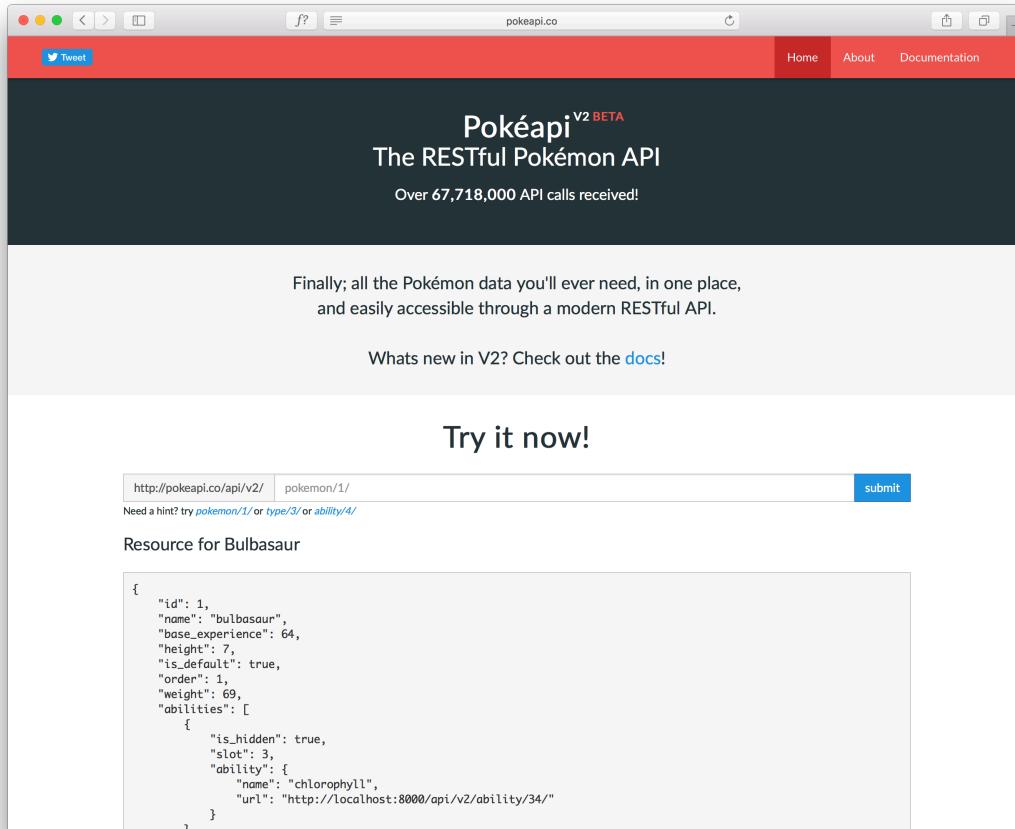


Abbildung 5.1: Die PokeAPI: *Finally; all the Pokémon data you'll ever need, in one place, and easily accessible through a modern RESTful API.*

Im Repository findet ihr unter dem Branch `pokedex` die Beispiel-Implementierung für die PokeAPI aus der Vorlesung (s. S. 32, Abb. 5.2). Ihr könnt das Repository klonen und mit

¹ `git checkout pokedex`

in den Branch wechseln und die App ausführen, oder den Branch auf GitHub [16] anschauen.

Wie in der Vorlesung am Beispiel der PokeAPI vorgeführt könnt ihr grob wie folgt vorgehen:

¹⁴<http://pokeapi.co>

¹⁵<https://swapi.co>

¹⁶<https://github.com/iOS-Dev-Kurs/apiclient/tree/pokedex>



Abbildung 5.2: Die Pokedex App zeigt Daten der PokeAPI an.

- Sucht euch eine API aus und überlegt euch eine passende Benutzeroberfläche. Fangt klein an und beschränkt euch zunächst auf nur einen oder wenige Endpoints der API! Pokemon-Attacken [17], Planeten im Star Wars Universum [18], Marvel Comic Charaktere [19]... oder lieber das Wetter [20]?

Endpoints mit wenigen Verweisen zu anderen Endpoints sind einfacher zu bearbeiten als bspw. Listen, die ihr für eine Table View dynamisch nachladen müsst. Beginnt mit einem Textfeld und einem *Laden* Button bevor ihr eure Benutzeroberfläche ausbaut.

- Modelliert dann die API Endpoints als Fälle eines Enums und implementiert das `Moya.TargetType` Protokoll:

```

1 import Moya
2 enum YourAPI: Moya.TargetType {
3     case firstEndpoint(someParameter: String, anotherParameter:
4         Int)
5     // ...

```

¹⁷<http://pokeapi.co/docs/v2/#moves>

¹⁸<https://swapi.co/documentation#planets>

¹⁹<http://developer.marvel.com/docs>

²⁰<https://developer.forecast.io/docs/v2>

```

5
6  var baseURL: NSURL { return NSURL(string:
7      ↵ "http://yourapi.com")! }
8  var path: String {
9      switch self {
10         case .firstEndpoint(someParameter: let a, _): return
11             ↵ "/firstEndpoint/\(a)"
12     }
13 }
14 var method: Moya.Method { return .GET }
15 var parameters: [String : AnyObject]? {
16     switch self {
17         case .firstEndpoint(_, anotherParameter: let b): return
18             [
19                 "anotherParameter": b,
20             ]
21     }
22 }
23 var sampleData: NSData { return
24     ↵ """.dataUsingEncoding(NSUTF8StringEncoding)! }
25 }
```

Die Modellierung der PokeAPI findet ihr hier [21].

3. Erstellt im App Delegate einen `MoyaProvider` für eure API und gebt ihn an euren ersten View Controller weiter.
4. Führt bei Betätigung eines Buttons oder in `viewDidLoad` testweise Requests aus:

```

1 // Where `api` is the `MoyaProvider`:
2 api.request(target) { result in
3     switch result {
4         case .Success(let response):
5             do {
6                 try response.filterSuccessfulStatusCodes()
7                 print(response)
8             } catch {
9                 print(error)
10            }
11        case .Failure(let error):
12            print(error)
13    }
14 }
```

²¹<https://github.com/iOS-Dev-Kurs/apiclient/blob/pokedex/APIClient/PokeAPI.swift>

Denkt daran, dass für ungesicherte HTTP Requests eine NSAppTransport-Security Ausnahme in der "["Info.plist"](#)" Datei eingetragen sein muss. Den Eintrag könnt ihr aus der "["Info.plist"](#)" Datei im pokedex Branch [22] kopieren.

5. Verwendet das Freddy Framework um den Rückgabewert der Request von [NSData](#) to JSON zu parsen:

```
1 import Freddy
2 let json = try JSON(data: response.data)
```

6. Modelliert nun eure Datenstrukturen. Schaut euch dazu die Dokumentation zu den API Endpoints an und implementiert Structs, die die Rückgabewerte der Endpoints genau repräsentieren. Implementiert das [Freddy.JSONDecodable](#) Protokoll für jedes Struct. Beschränkt euch zunächst nur auf die wichtigsten Attribute. Die Beispiel-Implementierung für die PokeAPI findet ihr hier [23] und hier [24].
7. Jetzt könnt ihr das Ergebnis einer Request in euer Model parsen und die View-Komponente eurer App damit konfigurieren:

```
1 let model = try Model(json: json) // Where `Model` is
   ↴ `Freddy.JSONDecodable`
2 // Configure view according to model
```

Modelliert mindestens einen Endpoint einer API eurer Wahl und zeigt das Ergebnis auf dem Bildschirm an, um die volle Punktzahl für diese Aufgabe zu erhalten. Mit komplexeren Implementierungen könnt ihr euch bis zu drei Extrapunkte verdienen. Schickt eine Pull-Request wenn ihr mit eurer App zufrieden seid, oder um Fragen zu stellen. Ich bin gespannt auf eure Umsetzungen!

²²<https://github.com/iOS-Dev-Kurs/apiclient/blob/pokedex/APIClient/Info.plist>

²³<https://github.com/iOS-Dev-Kurs/apiclient/blob/pokedex/APIClient/Pokedex.swift>

²⁴<https://github.com/iOS-Dev-Kurs/apiclient/blob/pokedex/APIClient/Pokemon.swift>

5.2 Unit Tests

In der Softwareentwicklung müssen wir natürlich kontinuierlich überprüfen, ob unsere Software funktioniert. Zunächst besteht so ein Test meist einfach darin, die App im Simulator oder auf einem Gerät auszuführen und auszuprobieren, ob sie sich wie erwartet verhält. Wird die App jedoch immer umfangreicher, können wir manuell nicht immer wieder alle möglichen Situationen testen, die in der App auftreten können. Teile der Software, die mal zuverlässig funktioniert haben, könnten dann durch neuen Code fehlerhaft werden, ohne, dass wir es bemerken.

Mit *Unit Tests* automatisieren wir daher solche Tests unserer Software. Für jedes Stück Code (jede *Unit Code*) das wir schreiben, fügen wir auch *Tests* hinzu, die diesen Code ausführen und das Ergebnis überprüfen. Diese Tests können wir dann jederzeit ausführen und damit sicherstellen, dass der getestete Code noch immer wie erwartet funktioniert. Wenn ein Test durchfällt erfahren wir außerdem genau, welcher Code fehlerhaft ist. Wir können das Problem dann beheben und die Tests erneut ausführen, bis alle Tests bestehen.

Je detaillierter und vielseitiger unsere Tests sind, desto genauer können wir feststellen, wie stabil unsere Software ist. Damit gehören Unit Tests zu den wichtigsten Werkzeugen moderner Softwareentwicklung, und Softwareprojekte wie die Open-Source Frameworks, die wir im vorherigen Abschnitt kennengelernt haben, werden nach dem Umfang ihrer Unit Tests (*Coverage*) bewertet [²⁵].

Da Unit Tests häufig und regelmäßig ausgeführt werden sollten, völlig automatisiert sind und durch ihren Umfang recht viel Zeit in Anspruch nehmen können werden sie häufig von einem *Continuous Integration System* wie *Travis*[²⁶] anstatt auf dem System des einzelnen Entwicklers ausgeführt. So können die Tests bspw. bei jedem *Push* auf GitHub und für jede Pull-Request automatisch ausgeführt werden um die Qualität und Stabilität des Codes zu bewahren.

²⁵<https://codecov.io/github/Moya/Moya?branch=master>

²⁶<https://travis-ci.org>

Übungsaufgaben

9. Unit Tests

[2 P.]

In dieser Aufgabe erweitern wir die *APIClient* App um Unit Tests. Wir schreiben dabei sowohl Tests, die möglichst detailliert unseren Programmcode ausführen und überprüfen, also auch UI Tests, die die App im Simulator ausführen und Benutzereingaben simulieren.

1. Im *APIClient* Repository [27] habe ich das Open-Source Framework *Nimble*²⁸] mit CocoaPods hinzugefügt und in zwei neue Test-Targets integriert. Fügt zuerst diese Änderungen am Original-Repository in euren Fork ein, indem ihr einen *Rebase* durchführt:

```
1 git pull --rebase https://github.com/ios-dev-kurs/apiclient.git
   ↳ master
```

Mit einem *Rebase* wird die Commitfolge verändert, daher müsst ihr euer Repository auf GitHub anschließend mit einer *Force Push* überschreiben:

```
1 git push --force origin master
```

2. Erstellt eine neue Datei "*YourAPITests.swift*" im "*APIClientTests*" Verzeichnis und achtet darauf, dass ihr sie dem Target *APIClientTests* zuweist (s. S. 37, Abb. 5.3). Erstellt darin eine neue Subklasse von *XCTestCase*:

```
1 import XCTest
2 // Import the App module with access to all internal types and
   ↳ functions
3 @testable import APIClient
4 // Nimble provides excellent testing functionality
5 import Nimble
6 // Every subclass of `XCTestCase` can provide tests
7 class YourAPITests: XCTestCase {
8     override func setUp() {
9         super.setUp()
10        // Called before the invocation of each test
11    }
12    override func tearDown() {
13        super.tearDown()
14        // Called after the invocation of each test
15    }
16 }
```

²⁷<https://github.com/iOS-Dev-Kurs/apiclient>

²⁸<https://github.com/Quick/Nimble>

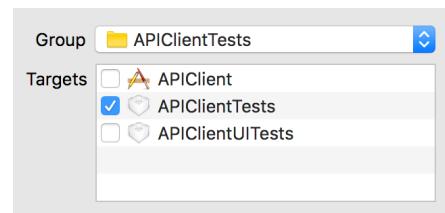


Abbildung 5.3: Weist Swift-Dateien mit Tests dem richtigen Target zu, wenn ihr die Datei erstellt.

3. Jetzt könnt ihr euren ersten Test schreiben. Jede Methode einer Subklasse von `XCTestCase` mit Signatur `() -> Void`, deren Name mit "test" beginnt, wird als Unit Test erkannt:

```

1 func testSomething() {
2     // Call some unit of code and check its result with
3     // → `XCTAssert` or Nimble's `expect` assertions:
4     expect(1 + 1) == 2
5     expect(try JSON(jsonString: "{}")).toNot(throwError())
}
```

Diese Methoden erscheinen dann im Xcode *Test Navigator* (und) und werden bei einem *Test* () automatisch ausgeführt. Ihr könnt sie mit der Schaltfläche am linken Rand des Editors auch einzeln ausführen.

Fügt eurem `XCTestCase` nun also Tests hinzu, mit denen ihr euren Code möglichst detailliert überprüft. Ihr könnt auch weitere Subklassen von `XCTestCase` erstellen, um eure Tests zu gliedern. Beginnt dabei bei der Model-Komponente eurer App und versucht Tests zu schreiben, die möglichst kleine Abschnitte eures Codes auszuführen, wie bspw. nur eine bestimmte Funktion oder einen Initializer. Überprüft dann das Ergebnis mithilfe der `XCTAssert` oder Nimble's `expect Assertions`^[29].

Anders als bei regulärem Programmcode könnt ihr in Tests beliebig langen und ausführlichen Code schreiben und solltet insbesondere den Test-Methoden möglichst deskriptive Namen geben. Achtet außerdem darauf, in euren Assertions keine Logik zu schreiben, sondern vergleicht ein Ergebnis direkt mit bestimmten Werten: Der Test `expect(1 + 1) == 2` im obigen Beispiel wäre sinnlos, würden wir den `+` Operator auch auf der rechten Seite verwenden!

Eine Beispiel-Implementierung einiger Tests findet ihr im `pokedex` Branch ^[30]. Es ist hilfreich, eine Datei mit zu testendem Code rechts im Assistant anzuzeigen (-Klick), dessen Funktionalität durchzugehen und dabei Tests

²⁹<https://github.com/Quick/Nimble>

³⁰<https://github.com/iOS-Dev-Kurs/apiclient/tree/pokedex>

dafür zu schreiben. Beginnt bspw. mit einem `JSONDecodable` Struct und dessen Initializer.

Um die `JSONDecodable` Initializer zu testen, ohne Netzwerkanfragen an die API stellen zu müssen, stellt das `Moya.TargetType` Protokoll das Attribut `sampleData: NSData` bereit, das ihr in eurem Enum vermutlich noch nicht vollständig implementiert habt. Gebt dort für jeden Endpoint einen JSON-String mit den relevanten Attributen zurück, der wie die echten API-Antworten strukturiert ist. Hinweis: Es gibt keine mehrzeiligen Strings in Swift und Anführungszeichen im String müsst ihr als "`\n`" escapen, wie hier [³¹].

Testet natürlich auch die asynchrone API Abfrage. Dazu können wir im Test eine *Expectation* erstellen und mit einem Timeout darauf warten, dass diese asynchron erfüllt wird:

```

1 func testSomethingAsynchronous() {
2     let expectation = expectationWithDescription("Something
3         ↪ asynchronous")
4     // Do something asynchronously and call
5         ↪ `expectation.fulfill()` when it's done.
6     waitForExpectationsWithTimeout(10, handler: nil)
7 }
```

4. Mit Unit Tests lässt sich die Model-Komponente ideal testen. Auch die View- und Controller-Komponenten können mit Unit Tests überprüft werden, wenn wir den Code dafür richtig strukturieren. Dabei hilft insbesondere, das Model-View-Controller Konzept zum *Model-View-ViewModel* Konzept zu erweitern [³²]. Doch erst mit *UI Tests* können wir unsere App auf einer Ebene testen, auf der wir tatsächlich Benutzereingaben simulieren und überprüfen, wie unsere App darauf reagiert.

UI Tests verwenden den *Accessibility* Mechanismus von iOS, der auch eingeschränkten Benutzern bei der Bedienung von Apps hilft. Dazu vergeben wir *Accessibility Identifier* an Views im Storyboard und im Code und können die zugehörigen Interface-Elemente damit in Tests überprüfen.

Erstellt eine neue Subklasse von `XCTestCase` und weist sie diesmal dem Target *APIClientUITests* zu. Startet die App in deren `setUp` Methode:

```

1 override func setUp() {
2     super.setUp()
3     continueAfterFailure = false
4     XCUIApplication().launch()
5 }
```

³¹<https://github.com/iOS-Dev-Kurs/apiclient/blob/pokedex/APIClient/PokeAPI.swift>

³²<https://www.objc.io/issues/13-architecture/mvvm/>

5. In Test-Methoden dieser Subklasse könnt ihr die angezeigten Interface-Elemente nun über ihren `accessibilityIdentifier` abfragen, Benutzereingaben simulieren und Asserts ausführen:

```

1 func testSomeUserInteraction() {
2     let app = XCUIApplication()
3     // Retrieve interface elements by their
4     // → `accessibilityIdentifier`
5     let searchTextField = app.textFields["searchTextField"]
6     // Assert on their properties such as `label` or `exists`
7     expect(searchTextField.label).to(beEmpty())
8     // Simulate user interaction
9     searchTextField.tap()
10    // Type something and hit enter
11    searchTextField.typeText("Search Term\n")
12    // Continuously assert on a property until an expectation is
13    // → met
14    let resultLabel = app.staticTexts["result"]
15    expect(resultLabel.label).toEventually(equal("Expected
        // Result"), timeout: 10)
16    // ...
17 }
```

Die Accessibility Identifier der Views müsst ihr dafür im Storyboard im *Identity Inspector* ( +  + ) oder im Code setzen. Für Views in Table View Cells scheint dies bisher nur im Code zu funktionieren:

```

1 // In UITableViewCell subclasses:
2 override func awakeFromNib() {
3     super.awakeFromNib()
4     self.nameLabel.accessibilityIdentifier = "name"
5 }
```

Um Text in Textfelder zu tippen muss die Option *Connect Hardware Keyboard* des Simulators ausgeschaltet sein (s. S. 40, Abb. 5.4). Außerdem scheint der UI Test beim Start der App bisher nicht korrekt zu warten, bis die App Benutzereingaben annimmt. Ein Workaround ist, die folgende Methode in `setUp` nach dem Start der App aufzurufen:

```

1 private func waitForResponsiveness() {
2     let wait = expectationWithDescription("wait")
3     dispatch_after(dispatch_time(DISPATCH_TIME_NOW, Int64(1 *
        // Double(NSEC_PER_SEC))), dis-
        // patch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND,
        // 0)) {
```

```

4           wait.fulfill()
5       }
6   waitForExpectationsWithTimeout(10, handler: nil)
7 }
```

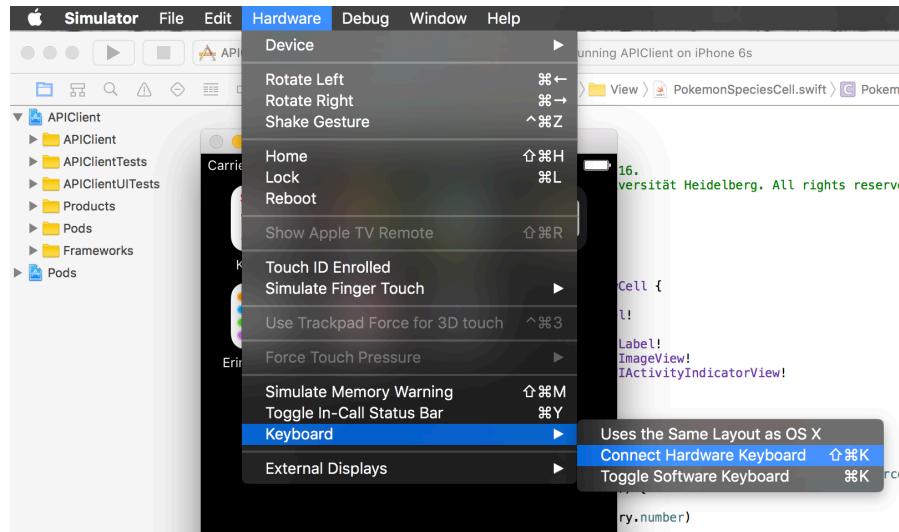


Abbildung 5.4: Nur mit ausgeschaltetem Hardware Keyboard funktioniert die simulierte Texteingabe bisher halbwegs zuverlässig.

Wenn ihr einen UI Test ausführt startet nun eure App und die Benutzereingaben werden simuliert. Wenn ihr innerhalb eines Tests mit Klick auf den linken Editorrand einen Breakpoint setzt und die Ausführung der App anschließend an der Stelle stoppt, könnt ihr sogar die *Recording*-Funktion verwenden, um eure Benutzereingaben aufzunehmen und automatisch in Code zu übersetzen! Klickt dazu auf die rote *Aufnehmen*-Schaltfläche in der Leiste über dem Debug-Bereich. Der generierte Code ist natürlich häufig nicht optimal und ihr solltet ihn noch editieren, bevor ihr den Test committed, doch diese Funktion kann sehr hilfreich sein.

Schreibt UI Tests für jeden Pfad der Benutzerführung in eurer App.

Ihr könnt eure Tests nun jederzeit mit $\text{⌘} + \text{U}$ ausführen und damit überprüfen, dass euer Code noch funktioniert.

Am rechten Rand des Editors zeigt euch Xcode dann außerdem Informationen zur *Coverage* an und markiert insbesondere solche Bereiche eures Codes rot, die in den Tests nicht aufgerufen wurden. Wählt ihr einen Test im *Report Navigator* ($\text{⌘} + \text{8}$) aus, seht ihr neben einer Übersicht der ausgeführten Tests auch die Coverage eures Projekts, einzelner Dateien und Funktionen. Eine hohe Coverage heißt nur, dass der Code während des Tests ausgeführt wurde, und nicht, wie detailliert die Tests sind!

Versucht, eine Coverage von mindestens 80% zu erreichen. Schickt dann erneut eine Pull-Request an das APIClient Repository.

5.3 Data Persistence

Selten kommt eine App ohne Daten aus, die über die Aufführung der App hinaus gespeichert werden. Insbesondere ist eines der wichtigsten Merkmale, die Apps von mobilen Webseiten unterscheidet, dass der Benutzer sie auf seinem Gerät personalisieren kann. Außerdem erwartet der Benutzer, dass er die App in dem Zustand wiederfindet, in der er sie verlassen hat. Da das Betriebssystem die App jedoch jederzeit beenden kann, muss die App Daten persistent speichern, um sie bei der nächsten Ausführung wiederherstellen zu können.

Um Daten zwischen Geräten desselben Benutzers zu teilen und damit ein konsistentes Benutzererlebnis zu ermöglichen, kann eine App Daten außerdem recht einfach nicht nur lokal sondern in der Cloud speichern.

Je nach Anforderung können wir auf verschiedene Mechanismen zur persistenten Datenspeicherung zurückgreifen:

5.3.1 User Defaults

Häufig erfordert die Personalisierung einer App lediglich die Speicherung einiger Einstellungen. Einfache Benutzerdaten können wir mithilfe der `NSUserDefaults` Klasse persistent speichern und auslesen:

```

1 // speichern
2 NSUserDefaults.standardUserDefaults().setObject("Alice",
  ↪ forKey:@"user_name")
3 // auslesen
4 let userName =
  ↪ UserDefaults.standardUserDefaults().objectForKey("user_name") as?
  ↪ String

```

Es können nur Objekte der Objective-C Typen `NSString` (implizit konvertierbar in Swift's `String`), `NSNumber` (implizit konvertierbar in Swift's `Int` und `Double`), `NSDate` und `NSData` oder ausschließlich mit solchen Objekten gefüllte Instanzen von `NSArray` und `NSDictionary` (bzw. ihre Swift-Äquivalente) in dieser Form gespeichert werden. Andere Datentypen müssen wir in `NSData` konvertieren, um sie in `NSUserDefaults` zu speichern. Für Klassen wie `UIImage` existieren dazu einfache Lösungen:

```

1 // `UIImage` in `NSData` serialisieren
2 let imageData = UIImagePNGRepresentation(image) // oder
  ↪ `UIImageJPEGRepresentation` je nach Bedarf
3 // speichern
4 UserDefaults.standardUserDefaults().setObject(imageData,
  ↪ forKey:@"image_data")
5 // auslesen

```

```

6 if let imageData =
  ↵ UserDefaults.standardUserDefaults().objectForKey("image_data") as?
  ↵ NSData {
7   let image = UIImage(data: imageData)
8 }
```

5.3.2 Caches

Für etwas komplexere Datentypen wie bspw. eigene Structs oder Klassen, oder Anforderungen wie ablaufende Caches, können wir auf spezialisierte Swift Frameworks zurückgreifen. Einige hervorragende Frameworks sind:

Pantry^[33] *The missing light persistence layer for Swift*

Haneke^[34] *A lightweight generic cache for iOS written in Swift with extra love for images.*

AwesomeCache^[35] *Delightful on-disk cache (written in Swift)*

5.3.3 State Preservation

Das Betriebssystem kann eine App jederzeit beenden. Der Benutzer der App erwartet dabei, dass die Benutzeroberfläche beim nächsten Start der App wiederhergestellt wird und Eingaben erhalten bleiben.

UIKit bietet mit *State Preservation* ein sehr einfach zu integrierendes System zur Wiederherstellung der Benutzeroberfläche. Konzeptionell sollten mit diesem Mechanismus keine Daten der Model-Komponente gespeichert werden. Stattdessen bezieht sich das State Preservation System auf die View- und Controller-Komponente zugeordnet.

Elemente der Benutzeroberfläche identifizieren wir anhand des `restorationIdentifier`: `String` Attributs im Storyboard oder im Code. Bei der Wiederherstellung wird für jeden dieser Identifier ein entsprechendes Objekt angefordert, oder bei Verwendung eines Storyboards automatisch erstellt. View Controller und Views können dazu die `encodeRestorableStateWithCoder(_:_)` und `decodeRestorableStateWithCoder(_:_)` Methoden implementieren, um ihre Darstellung zu archivieren und wiederherzustellen. Im iOS App Programming Guide [36] ist das State Preservation System ausführlich dokumentiert.

5.3.4 Datenbanken: Core Data und Realm

Gehen die Anforderungen an die persistente Datenspeicherung unserer Model-Komponente über einfaches Caching einzelner Werte hinaus, benötigen wir eine relationale Datenbank. Momentan bietet sich uns die Wahl zwischen Apple's etabliertem *Core Data* Framework und dem modernen Framework *Realm*^[37]. Core Data ist ein Relikt aus jahrelanger

³⁶<https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/>

³⁷<https://realm.io>

Softwareentwicklung auf Apple's Plattformen in Objective-C und damit sehr mächtig und vielseitig und liegt einer Vielzahl an Apps im Store zugrunde. Realm hingegen ist eine plattformübergreifende Datenbankimplementierung für mobile Geräte mit expliziter Unterstützung für Swift, jedoch noch nicht so vollständig und in Apple's Infrastruktur integriert wie Core Data.

Beide Frameworks erlauben uns mit Swift Objekten zu arbeiten, denen eine Datenbank zugrunde liegt. Wir erstellen Objekte einer Subklasse von `NSManagedObject` bzw. `Realm.Object` und arbeiten mit diesen Objekten, wie wir es gewohnt sind. Als `@NSManaged` bzw. `dynamic` markierte Attribute dieser Klassen werden in der Datenbank gespeichert. Dazu gehören auch *To-One* und *To-Many* Beziehungen zu anderen verwalteten Objekten. So können wir ganze Objekt-Graphen von Core Data bzw. Realm persistent verwalten lassen.

Mit **Core Data** modellieren wir mit dem *Xcode Model Editor* zuerst das *Data Model* der Datenbank (s. S. 44, Abb. 5.5) mit *Entities* und ihren *Attributen* und *Beziehungen*. Anschließend implementieren wir eine Subklassen von `NSManagedObject` für jede Entity.

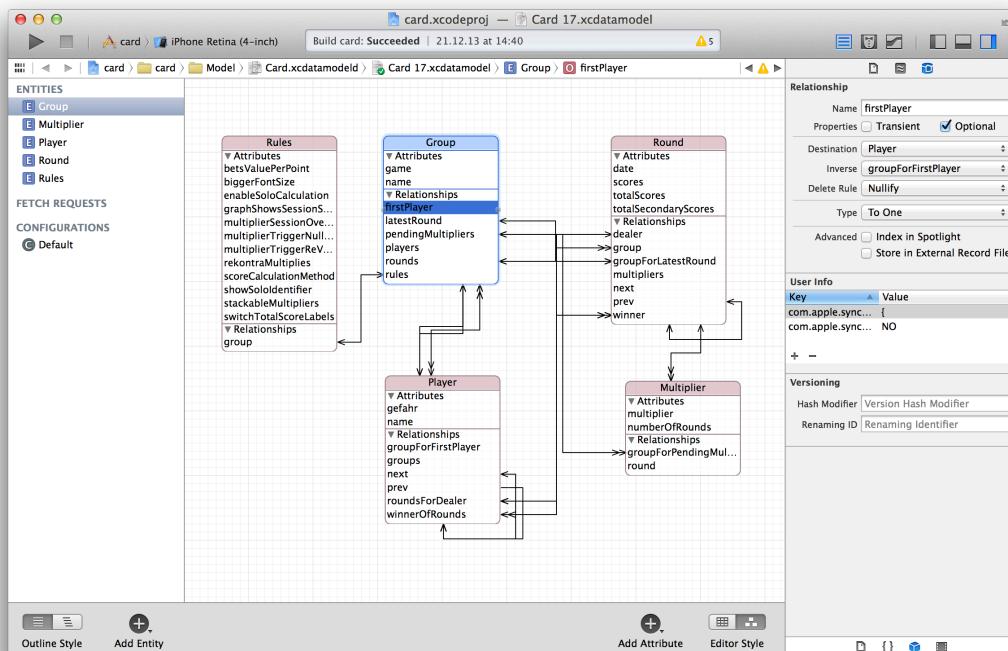


Abbildung 5.5: Core Data Models können wir im *Xcode Model Editor* graphisch bearbeiten.

Um von Core Data verwaltete Objekte abzufragen führen wir eine *FetchRequest* aus. Wir konfigurieren dazu ein Objekt der Klasse `NSEntityDescription` und übergeben es an einen `NSManagedObjectContext`.

Ein `NSManagedObjectContext` ist unsere Verbindung zu Core Data. Meist erstellen wir einen Context im App Delegate und geben ihn an unsere View Controller weiter, sodass

diese die von Core Data verwalteten Objekte abfragen können. Diese Objekte sind dann immer mit dem Context assoziiert. Wenn wir ihre Attribute verändern entspricht dies ungesicherten Änderungen in ihrem Context. Erst, wenn wir den Context speichern, werden die Änderungen persistent gesichert.

Core Data stellt viele mächtige Hilfsmittel bereit, um mit verwalteten Objekten zu arbeiten. Insbesondere der `NSFetchedResultsController` hilft, ein Subset von verwalteten Objekten in einer Table View darzustellen. Wir erstellen einen `NSFetchedResultsController` mit einer Fetch Request und führen einmal dessen `performFetch` Methode aus. In der Implementierung der `UITableViewDatasource` können wir dann auf die genau darauf ausgelegten Attribute des `NSFetchedResultsController` zurückgreifen. Außerdem reagiert der `NSFetchedResultsController` auf Änderungen der Objekte und kann die Table View danach anpassen.

Übungsaufgaben

10. Data Persistence [2 P.]

Forkt das *DataPersistence* Repository [38] und implementiert eine einfache App, die das Core Data Framework verwendet um relationale Daten in einer Datenbank zu speichern und anzuzeigen.

Implementiert eine *Todo-Liste* wie im Beispiel unter dem Branch `todo`, oder etwas anderes.

1. Definiert zuerst das *Data Model* im Xcode Model Editor. Öffnet dazu die `".xcdatamodeld"` Datei und erstellt die Entities, Attribute und Beziehungen, die euer Model benötigt.
2. Erstellt dann für jede Entity eine Swift-Datei, in der ihr eine Subklasse von `NSManagedObject` mit den als `@NSManaged` markierten Attributen eures Models implementiert. Vergesst nicht, den Namen der Klasse anschließend als Identität der Entity im Data Model zu setzen und darunter als Modul `"Current Product Module"` einzugeben.
3. Schaut euch an, wie im App Delegate und in der `PersistentStack` Klasse ein Core Data Stack und ein zugehöriger `NSManagedObjectContext` erstellt wird. Gebt den `persistentStack.mainContext` an euren ersten View Controller weiter, sodass dieser einen Zugang zur Datenbank erhält.
4. Implementiert eure View Controller. Verwendet `NSFetchedResultsController` mit `UITableViews`, wie in der Todo-Liste vorgeführt, um Listen von Objekten aus der Datenbank anzuzeigen.

Erstellt für View Controller, die Objekte erstellen oder editieren, einen neuen `NSManagedObjectContext`, dessen `parentContext` ihr auf den `mainContext` setzt. Gebt diesen Context dann an den View Controller weiter, der darin speichern kann. Änderungen, die in einem solchen *Child Context* gespeichert werden, erscheinen dann im *Parent Context* als ungesicherte Änderungen. Erst wenn ihr den Parent Context anschließend auch speichert werden die Änderungen in die Datenbank gesichert. Bei Bedarf, bspw. bei Betätigung eines *Cancel*-Buttons, könnt ihr den Child Context dann einfach nicht speichern. Orientiert euch daran, wie dies bspw. in den Segues und Unwind Segues des `ListsViewController` und `CreateListViewController` der Todo-Liste implementiert ist.

Schickt eine Pull-Request wenn ihr mit eurer App zufrieden seid, oder um Fragen zu stellen.

³⁸ <https://github.com/ios-dev-kurs/datapersistence>