# Neues Swift-Feature Async/Await

Auf der diesjährigen Swift Heroes Konferenz (April 2021) wurde in einem Vortrag von Vincent Pradeilles das neue Swift Feature Async/Await (ab Swift 5.5) vorgestellt. Wir iOS Entwickler aus dem Thalia App Team haben uns den Vortrag angehört und waren begeistert von den neuen Möglichkeiten des Sprachfeatures, zur Vereinfachung von asynchronem Code. Andere Programmiersprachen (wie zB. C#) besitzen dieses Feature schon länger und wir sind froh, dass Swift jetzt nachgezogen hat.

Viele iOS Entwickler kennen das Problem, dass asynchroner Code schnell unleserlich, unübersichtlich und kompliziert werden kann. Genau dieses Problem soll Async/Await lösen, weshalb wir die Funktionsweise und Benutzung im folgenden Artikel mit Codebeispielen veranschaulichen wollen. Dabei stellen wir Async/Await auch den Alternativen gegenüber, um die Vorteile und Unterschiede zu bisherigen Verfahren deutlich zu machen.

Ziel des Artikels soll es sein, für Entwickler, die noch keine Berührung mit dem neuen Swift-Sprachfeature Async/Await hatten, die Grundlagen des Features zu erläutern. Für diejenigen, die sich schon ein bisschen mit dem Thema beschäftigt haben, empfehlen wir zur Vertiefung des Wissens die WWDC 2021 Session über Async/Await anzuschauen.

# Problemstellung und Lösungsmöglichkeit mit Completions

In dem Beispiel aus dem Vortrag von Vincent Pradeilles soll der Anwender mit Vor- und Nachnamen begrüßt werden. (Konsolenausgabe) Dafür muss eine UserId geladen werden, um damit den Vor- und Nachnamen zu laden. Die klassische Lösungsmöglichkeit für dieses Problem in Swift sind Completions.

In Abb. 1 sehen wir, wie eine solche Problemlösung mit Completions aussehen könnte. Wir sehen das der Code nicht so gut leserlich ist, da er eine Verschachtelung von Methodenaufrufen enthält (Pyramid of Doom), was bei diesem Beispiel noch einigermaßen verständlich ist, aber bei mehr Aufrufen schnell sehr unleserlich und kompliziert werden kann.

```swift
func getUserData() {
    getUserId { userId in
        getUserFirstname(userId: userId) { firstname in
            getUserLastname(userId: userId) { lastname in
                print("Hello \(firstname) \(lastname)")
            }
        }
    }
}
```

Abb. 1: Asynchroner Swift Code mit Completions

# Implementierung mit Async/Await

```swift
func greetUser() async {
    let userId = await getUserId()
    let firstname = await getUserFirstname(userId: userId)
    let lastname = await getUserLastname(userId: userId)

    print("Hello \(firstname) \(lastname)")
}
```

Abb. 2: Asynchroner Swift Code mit Async/Await

In Abb. 2 sehen wir, wie das Problem mit Async/Await gelöst werden kann. Das async-Keyword dient dazu dem Compiler zu signalisieren, dass in dieser Methode asynchrone Aufrufe stattfinden. Das await-Keyword bedeutet, dass hier auf das Ergebnis eines asynchronen Aufrufes gewartet werden muss. Im ersten Schritt wird über die Methode **getUserId()** die **userId** geladen. Erst wenn die **userId** geladen wurde, wird über die Methode **getUserFirstname** der **firstname** geladen. Im letzten Schritt wird der **lastname** geladen. D.h. die Aufrufe erfolgen **seriell** nacheinander.

**Achtung:** Enthält eine Methode ein await-Keyword, muss diese auch als asynchron über das async-Keyword markiert werden.

In Abb. 3 ist ein Beispiel zu sehen, um innerhalb der asynchronen Methode **getUserId()** eine Methode mit einer Completion aufzurufen (zB. um abwärtskompatibel zu bestehendem Code zu sein). Dazu kann man die Methode **withCheckedContinuation** aus der Swift Standard Library verwenden. Dabei wird die Rückgabe der UserId erst ausgeführt, wenn die **resume** Methode

auf dem Continuation-Objekt aufgerufen wird.

```
func getUserId() async -> Int {
    return await withCheckedContinuation({ continuation in
        getUserId { userId in
            continuation.resume(returning: userId)
        }
    })
}
```

Abb. 3: Aufruf einer Methode mit Completion aus einer mit async markierten Methode

In vielen Fällen ist es möglich asynchrone Methoden **parallel** auszuführen, um Zeit zu sparen. Mit Async/Await ist dies sehr einfach umzusetzen, durch Verwendung der Keywords **async let** wie in Abb. 4 zu sehen:

```
func greetUser() async {
    let userId = await getUserId()
    async let firstname = await getUserFirstname(userId: userId)
    async let lastname = await getUserLastname(userId: userId)

    await print("Hello \(firstname) \(lastname)")
}
```

Abb. 4: Parallele asynchrone Methodenaufrufe mit Async/Await

Zunächst wird, wie im ursprünglichen Code, die **userId** geladen. Durch die Deklaration der beiden Variablen **firstname** und **lastname** mit **async let** werden die beiden Methoden **getUserFirstname** und **getUserLastname** parallel ausgeführt. Erst wenn beide Methoden einen Rückgabewert liefern, wird das print-Statement ausgeführt. Dies wird erreicht durch das **await-Keyword** vor dem print-Statement.

# Lösung mit Combine

Eine andere Lösungsmöglichkeit in Swift ist Combine zu nutzen.

In Abb. 5 sehen wir, wie das Problem mit Combine gelöst werden kann.

```swift
func greetUser() {
    _  = getUserId()
    .flatMap { userId in
        return getUserFirstname(userId: userId).zip(getUserLastname(userId: userId))
    }.sink(receiveValue: { firstname, lastname in
        print("Hello using Combine \(firstname) \(lastname)")
    })
}
```

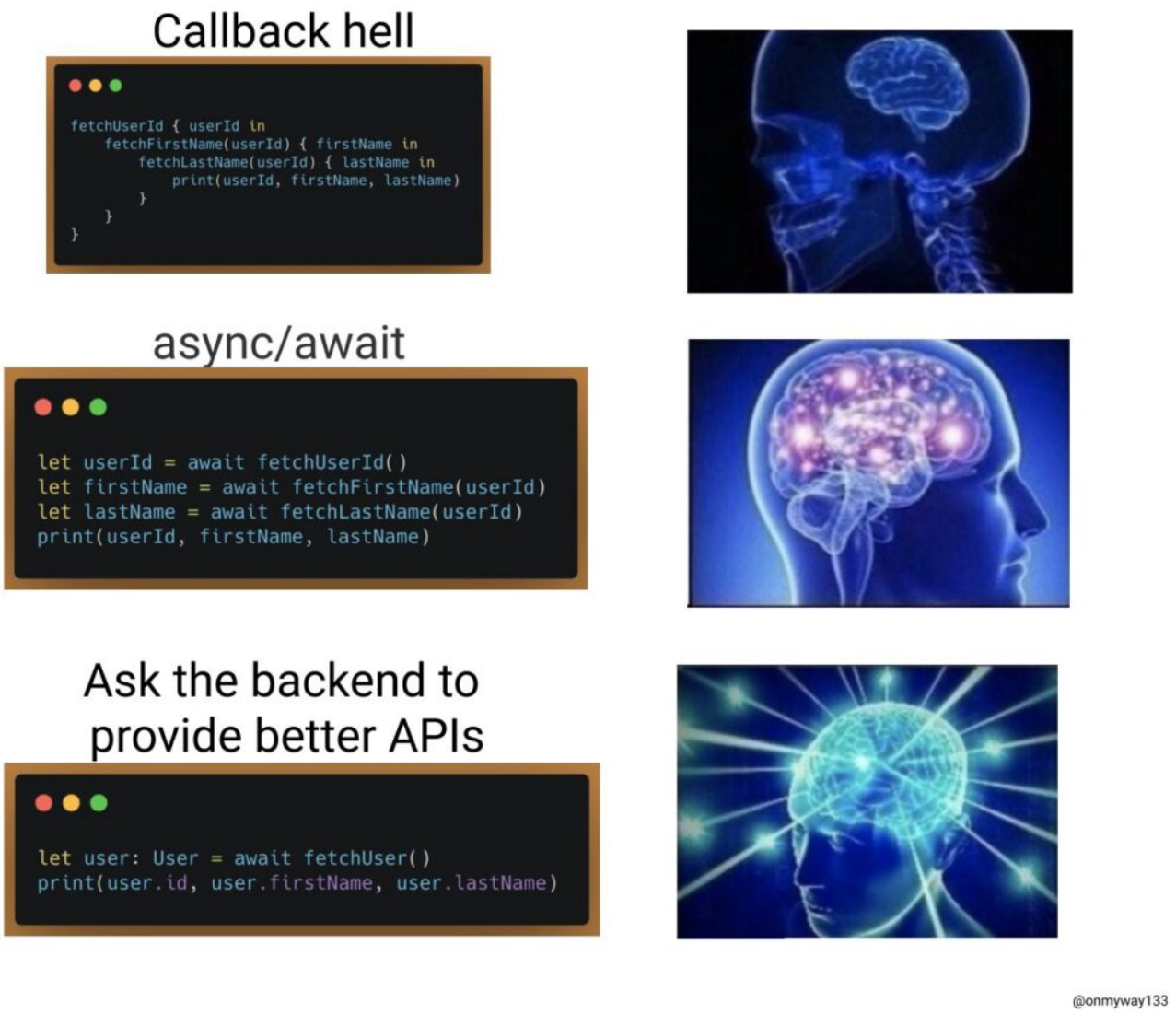
Abb. 5: Asynchroner Swift Code mit Combine

Die Lösung mit Combine ist nicht so tief verschachtelt wie die klassiche Lösung mit Completions, allerdings für diesen Anwendungsfall auch nicht sehr leserlich und etwas kompliziert.

# Was ist nun die bessere Lösung?

Die Frage lässt sich nicht so leicht beantworten. Combine ist ein Werkzeug, das viel mehr bietet als Async/Await. Allerdings löst Async/Await dieses spezielles Problem besonders gut. Je nach Anwendungsfall muss entschieden werden, ob eine Lösung mit Combine oder Async/Await die bessere Wahl darstellt.

# Lösung mit besseren Server APIs 




Quelle: Twitter ([https://twitter.com/onmyway133/status/1407772929031651328?s=21](https://twitter.com/onmyway133/status/1407772929031651328?s=21))

# Fazit

Wir haben gesehen, wie man mit Async/Await Code schreiben kann, der klassichen Code mit Completions ersetzt. Außerdem wie man Code mit Completions aus einer async-Methode aufrufen kann (Abwärtskompatibilität) und wie man mit async let mehrere Aufrufe parallel ausführen kann. Es wurde auch gezeigt das das Problem mit Combine lösbar ist.

Uns hat der Vortrag von Vincent Pradeilles auf der Swift Heroes 2021 und die WWDC 2021 Sessions zu Async/Await so gut gefallen, dass wir das Sprachfeature gerne in Zukunft in der Thalia App verwenden möchten. Dabei ist zu beachten, dass die Verwendung erst ab iOS 15 möglich ist.

Natürlich sind die anderen Lösungswege über Combine und Completions genauso valide und es muss von Fall zu Fall unterschieden werden, welche Lösung die Beste für das jeweilige Problem darstellt.