SwiftUI, async / await, AsyncAlgorithms: how does it fit?

Vincent Pradeilles (@v_pradeilles) – PhotoRoom

Do you remember WWDC 2021?


APPLE
IOS DEVELOPER
WE ADDED
ASYNC/AWAIT TO SWIFT
AND IT WON'T
REQUIRE IOS 15, RIGHT?
IT WON'T
REQUIRE IOS 15, RIGHT?
imgflip.com

But then, in December 2021…

John Sundell
@johnsundell

Amazing news, everyone! The new Swift concurrency system is now backward compatible all the way back to iOS 13, macOS Catalina, watchOS 6, and tvOS 13! 🎉😀🚀

Huge thanks to everyone at Apple and in the Swift open source community who made this happen! 🙌

Traduire le Tweet

Swift

New Features

- You can now use Swift Concurrency in applications that deploy to macOS 10.15, iOS 13, tvOS 13, and watchOS 6 or newer. This support includes async/await, actors, global actors, structured concurrency, and the task APIs. (70738378)

Now is a perfect time to start using async / await with SwiftUI 🔥

Let’s take a look at a simplified networking and model layer

simplified networking and model layer

```
struct Movie: Decodable, Equatable, Identifiable {
    let id: Int
    let title: String
    let overview: String
}

struct MovieResponse: Decodable {
    let results: [Movie]
}

let apiKey = "redacted 🙈"
```

simplified networking and model layer

```
func loadMovies(page: Int = 1) async -> [Movie] {
    do {
        let url = URL(string: "https://api.themoviedb.org/3/movie/upcoming?
api_key=\(apiKey)&language=en-US&page=\(page)")!

        let (data, _) = try await URLSession.shared.data(from: url)

        let decoder = JSONDecoder()

        let decoded = try decoder.decode(MovieResponse.self, from: data)

        return decoded.results
    } catch {
        return []
    }
}
```

simplified networking and model layer

```
func loadMovies(page: Int = 1) async -> [Movie] {
    do {
        let url = URL(string: "https://api.themoviedb.org/3/movie/upcoming?
api_key=\(apiKey)&language=en-US&page=\(page)")!

        let (data, _) = try await URLSession.shared.data(from: url)

        let decoder = JSONDecoder()

        let decoded = try decoder.decode(MovieResponse.self, from: data)

        return decoded.results
    } catch {
        return []
    }
}
```

simplified networking and model layer

```
func loadMovies(page: Int = 1) async -> [Movie] {
    do {
        let url = URL(string: "https://api.themoviedb.org/3/movie/upcoming?
api_key=\(apiKey)&language=en-US&page=\(page)")!

        let (data, _) = try await URLSession.shared.data(from: url)

        let decoder = JSONDecoder()

        let decoded = try decoder.decode(MovieResponse.self, from: data)

        return decoded.results
    } catch {
        return []
    }
}
```

simplified networking and model layer

```
func loadMovies(page: Int = 1) async -> [Movie] {
    do {
        let url = URL(string: "https://api.themoviedb.org/3/movie/upcoming?
api_key=\(apiKey)&language=en-US&page=\(page)")!

        let (data, _) = try await URLSession.shared.data(from: url)

        let decoder = JSONDecoder()

        let decoded = try decoder.decode(MovieResponse.self, from: data)

        return decoded.results
    } catch {
        return []
    }
}
```

simplified networking and model layer

```
func loadMovies(page: Int = 1) async -> [Movie] {
    do {
        let url = URL(string: "https://api.themoviedb.org/3/movie/upcoming?
api_key=\(apiKey)&language=en-US&page=\(page)")!

        let (data, _) = try await URLSession.shared.data(from: url)

        let decoder = JSONDecoder()

        let decoded = try decoder.decode(MovieResponse.self, from: data)

        return decoded.results
    } catch {
        return []
    }
}
```

simplified networking and model layer

```
func loadMovies(page: Int = 1) async -> [Movie] {
    do {
        let url = URL(string: "https://api.themoviedb.org/3/movie/upcoming?
api_key=\(apiKey)&language=en-US&page=\(page)")!

        let (data, _) = try await URLSession.shared.data(from: url)

        let decoder = JSONDecoder()

        let decoded = try decoder.decode(MovieResponse.self, from: data)

        return decoded.results
    } catch {
        return []
    }
}
```

simplified networking and model layer

```
func loadMovies(page: Int = 1) async -> [Movie] {
    do {
        let url = URL(string: "https://api.themoviedb.org/3/movie/upcoming?
api_key=\(apiKey)&language=en-US&page=\(page)")!

        let (data, _) = try await URLSession.shared.data(from: url)

        let decoder = JSONDecoder()

        let decoded = try decoder.decode(MovieResponse.self, from: data)

        return decoded.results
    } catch {
        return []
    }
}
```

simplified networking and model layer

```
func loadMovies(page: Int = 1) async -> [Movie] {
    do {
        let url = URL(string: "https://api.themoviedb.org/3/movie/upcoming?
api_key=\(apiKey)&language=en-US&page=\(page)")!

        let (data, _) = try await URLSession.shared.data(from: url)

        let decoder = JSONDecoder()

        let decoded = try decoder.decode(MovieResponse.self, from: data)

        return decoded.results
    } catch {
        return []
    }
}
```

Now building the view

building the view

```
struct ContentView: View {

    @State var movies = [Movie]()

    var body: some View {
        List(movies) { movie in
            HStack(spacing: 10) {
                VStack(alignment: .leading, spacing: 10) {
                    Text(movie.title)
                        .font(.headline)
                    Text(movie.overview)
                        .lineLimit(3)
                        .font(.subheadline)
                }
            }
        }
    }
}
```

How do we call `loadMovies(page:)`?

building the view

```
struct ContentView: View {

    @State var movies = [Movie]()

    var body: some View {
        List(movies) { movie in
            HStack(spacing: 10) {
                VStack(alignment: .leading, spacing: 10) {
                    Text(movie.title)
                        .font(.headline)
                    Text(movie.overview)
                        .lineLimit(3)
                        .font(.subheadline)
                }
            }
        }
    }
}
```

building the view

```
struct ContentView: View {

    @State var movies = [Movie]()

    var body: some View {
        List(movies) { movie in
            HStack(spacing: 10) {
                VStack(alignment: .leading, spacing: 10) {
                    Text(movie.title)
                        .font(.headline)
                    Text(movie.overview)
                        .lineLimit(3)
                        .font(.subheadline)
                }
            }
        }
        .onAppear {  Invalid conversion from 'async' function of type '() async -> ()' to synchronous function type '() -> Void'
            movies = await loadMovies()
        }
    }
}
```

building the view

```
struct ContentView: View {

    @State var movies = [Movie]()

    var body: some View {
        List(movies) { movie in
            HStack(spacing: 10) {
                VStack(alignment: .leading, spacing: 10) {
                    Text(movie.title)
                        .font(.headline)
                    Text(movie.overview)
                        .lineLimit(3)
                        .font(.subheadline)
                }
            }
        }
        .task { ✅
            movies = await loadMovies()
        }
    }
}
```

The `.task { }` modifier is the `async` equivalent of `.onAppear { }`

It automatically manages the lifetime of the `Task`, tying it to that of the `View`

Now let’s implement an infinite scroll

infinite scroll

```
struct ContentView: View {

    @State var movies = [Movie]()
    @State var currentPage = 1

    var body: some View {
        List(movies) { movie in
            HStack(spacing: 10) {
                VStack(alignment: .leading, spacing: 10) {
                    Text(movie.title)
                        .font(.headline)
                    Text(movie.overview)
                        .lineLimit(3)
                        .font(.subheadline)
                }
            }
            .task {
                if movie == movies.last {
                    currentPage += 1
                    movies += await loadMovies(page: currentPage)
                }
            }
        }
        .task {
            movies = await loadMovies()
        }
    }
}
```

infinite scroll

```
struct ContentView: View {

    @State var movies = [Movie]()
    @State var currentPage = 1

    var body: some View {
        List(movies) { movie in
            HStack(spacing: 10) {
                VStack(alignment: .leading, spacing: 10) {
                    Text(movie.title)
                        .font(.headline)
                    Text(movie.overview)
                        .lineLimit(3)
                        .font(.subheadline)
                }
            }
            .task {
                if movie == movies.last {
                    currentPage += 1
                    movies += await loadMovies(page: currentPage)
                }
            }
        }
        .task {
            movies = await loadMovies()
        }
    }
}
```

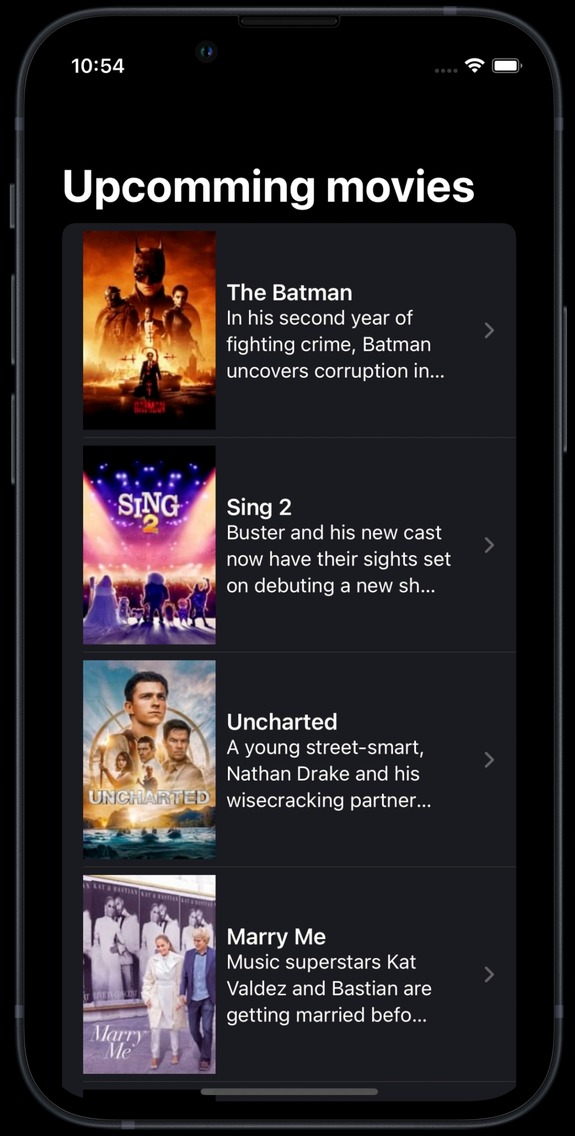
Let’s go one step further: Implementing concurrent calls!

Implementing concurrent calls

I want to implement a screen that displays both the cast and the reviews.

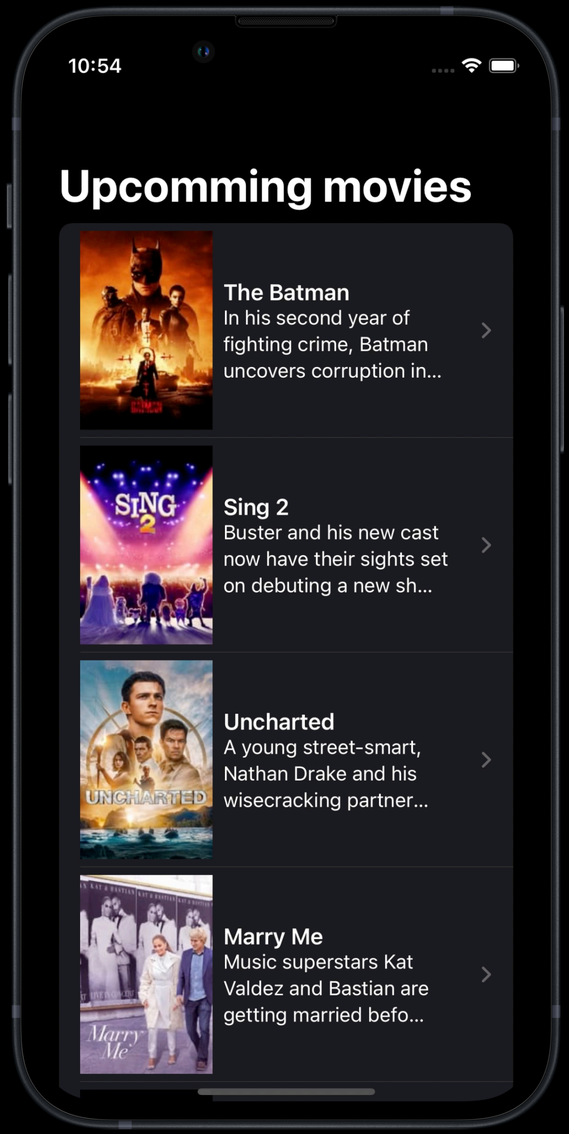
I'll need to fetch data from two separate endpoints.

Let's see how we can make it as efficiently as possible!



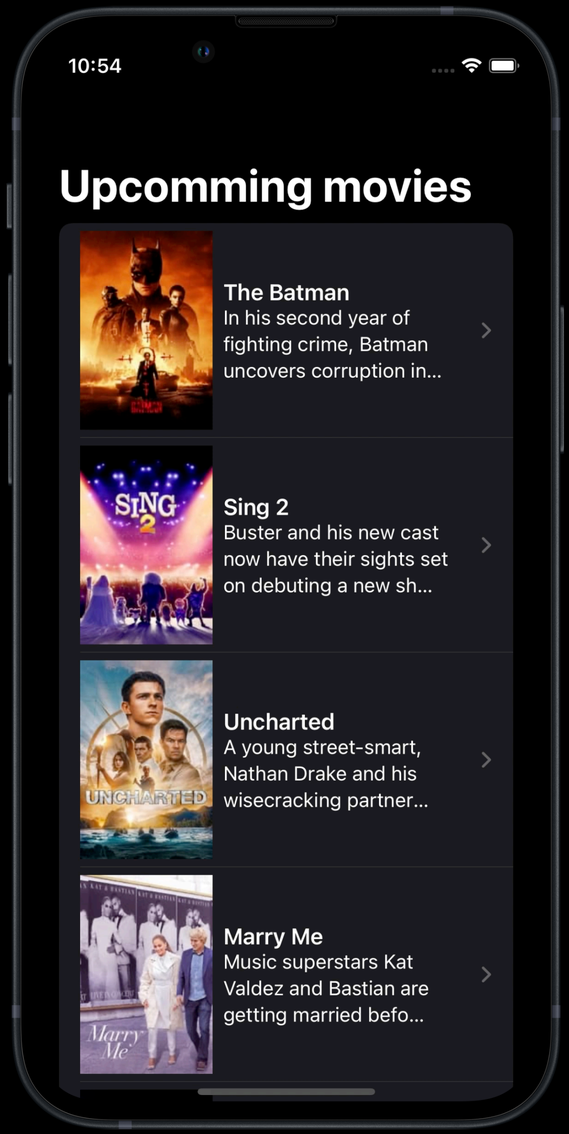
Implementing concurrent calls

```swift
import SwiftUI

struct DetailView: View {

    let movie: Movie

    @State var data: (credits: [MovieCastMember],
                      reviews: [MovieReview]) = ([], [])

    var body: some View {
        List {
            Section(header: Text("Credits")) {
                ForEach(data.credits) { credit in
                    VStack(alignment: .leading) {
                        Text(credit.name)
                            .font(.headline)
                        Text(credit.character)
                            .font(.caption)
                    }
                }
            }

            Section(header: Text("Reviews")) {
                ForEach(data.reviews) { review in
                    VStack(alignment: .leading, spacing: 8) {
                        Text(review.author)
                            .font(.headline)
                        Text(review.content)
                            .font(.body)
                    }
                }
            }
        }
        .navigationBarTitle(movie.title)
        .task {
            let credits = await getCredits(for: movie)
            let reviews = await getReviews(for: movie)

            data = (credits.cast, reviews.results)
        }
    }
}
```



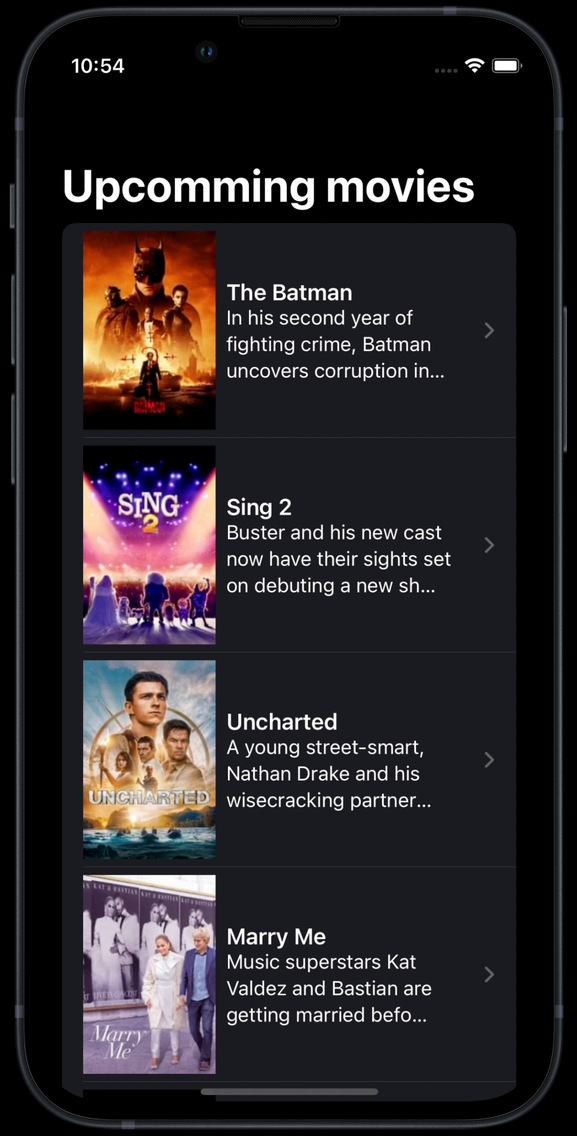
Implementing concurrent calls

```
var body: some View {
        List {
            Section(header: Text("Credits")) {
                ForEach(data.credits) { credit in
                    VStack(alignment: .leading) {
                        Text(credit.name)
                            .font(.headline)
                        Text(credit.character)
                            .font(.caption)
                    }
                }
            }

            Section(header: Text("Reviews")) {
                ForEach(data.reviews) { review in
                    VStack(alignment: .leading, spacing: 8) {
                        Text(review.author)
                            .font(.headline)
                        Text(review.content)
                            .font(.body)
                    }
                }
            }
        }
```



Implementing concurrent calls

```
.task {
    let credits = await getCredits(for: movie)
    let reviews = await getReviews(for: movie)

    data = (credits.cast, reviews.results)
}
```

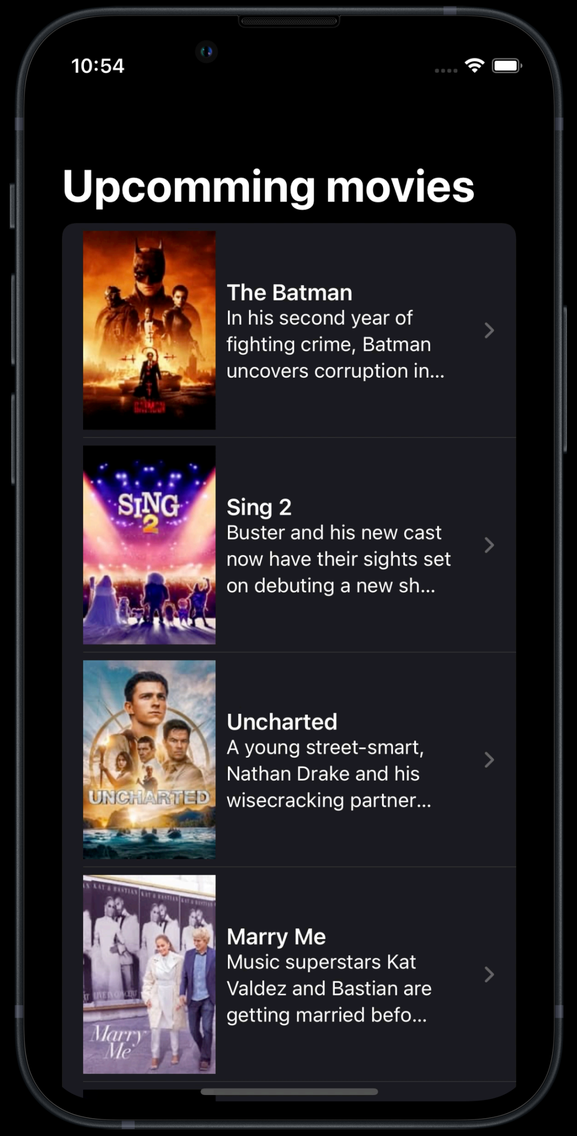


It works, but the calls are executed sequentially.

Implementing concurrent calls

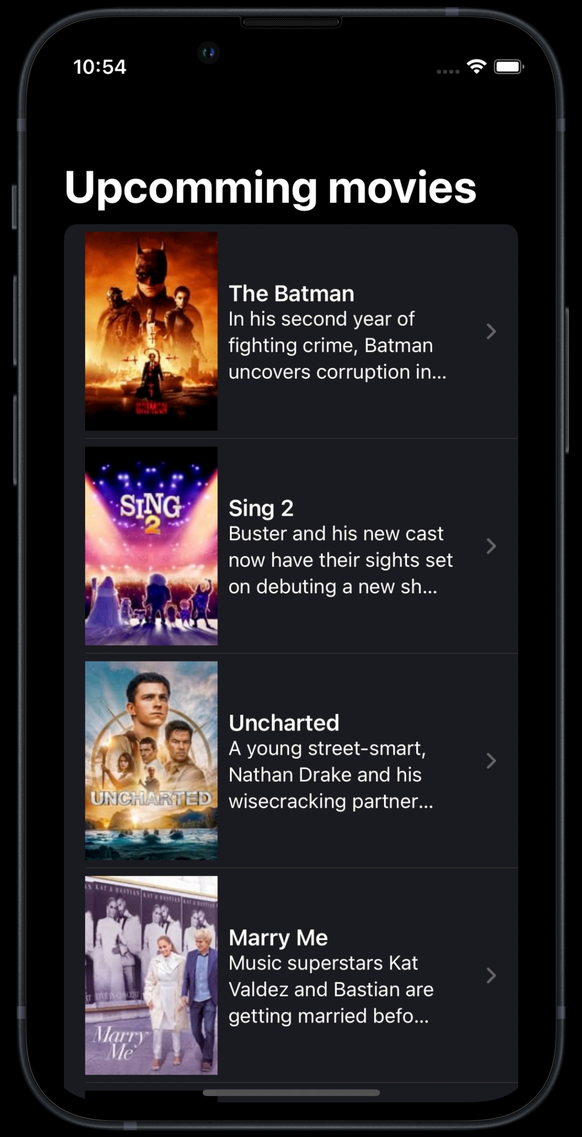
```
.task {
    async let credits = getCredits(for: movie)
    async let reviews = getReviews(for: movie)

    data = await (credits.cast, reviews.results)
}
```

Now both calls are happening concurrently 🔥



Implementing concurrent calls

```
.task {
    async let credits = getCredits(for: movie)
    async let reviews = getReviews(for: movie)

    data = await (credits.cast, reviews.results)
}
```



When you need to perform a static number of concurrent calls, **`async let`** is an amazing tool ✨

Where does Combine fit in all this?

Where does Combine fit in all this?

- Combine used to be Apple's first-party solution to deal with async code
- However, for simple use-cases `async` / `await` is a much better tool
- Still a few months ago, I would have argued that Combine was still the default tool for more complex use cases…
- …however

Introducing Swift Async Algorithms

MARCH 24, 2022 Tony Parker

> Tony Parker manages teams at Apple working on Foundation and Swift packages.

As part of Swift's move toward safe, simple, and performant asynchronous programming, we are pleased to introduce a new package of algorithms for `AsyncSequence`. It is called **Swift Async Algorithms** and it is available now on GitHub.

This package has three main goals:

- First-class integration with `async/await`
- Provide a home for time-based algorithms
- Be cross-platform and open source

Motivation

AsyncAlgorithms is a package for algorithms that work with *values over time*. That includes those primarily about *time*, like `debounce` and `throttle`, but also algorithms about *order* like `combineLatest` and `merge`. Operations that work with multiple inputs (like `zip` does on `Sequence`) can be surprisingly complex to implement, with subtle behaviors and many edge cases to consider. A shared package can get these details correct, with extensive testing and documentation, for the benefit of all Swift apps.

https://www.swift.org/blog/swift-async-algorithms/

Motivation

AsyncAlgorithms is a package for algorithms that work with *values over time*. That includes those primarily about *time*, like `debounce` and `throttle`, but also algorithms about *order* like `combineLatest` and `merge`. Operations that work with multiple inputs (like `zip` does on `Sequence`) can be surprisingly complex to implement, with subtle behaviors and many edge cases to consider. A shared package can get these details correct, with extensive testing and documentation, for the benefit of all Swift apps.

Motivation

AsyncAlgorithms is a package for algorithms that work with *values over time*. That includes those primarily about *time*, like `debounce` and `throttle`, but also algorithms about *order* like

Combine

Customize handling of asynchronous events by combining event-processing operators.

Overview

The Combine framework provides a declarative Swift API for processing values over time. These values can represent many kinds of asynchronous events. Combine declares *publishers* to expose values that can change over time, and *subscribers* to receive those values from the

So how does Combine code translate to AsyncAlgorithms? 😐

```swift
class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Publishers.CombineLatest3($userName, $password, $passwordConfirmation)
            .map { userName, password, passwordConfirmation in
                return userName.isEmpty == false
                    && password.isEmpty == false
                    && password == passwordConfirmation
            }
            .assign(to: &$canLogin)
    }
}
```

```
class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Publishers.CombineLatest3($userName, $password, $passwordConfirmation)
            .map { userName, password, passwordConfirmation in
                return userName.isEmpty == false
                    && password.isEmpty == false
                    && password == passwordConfirmation
            }
            .assign(to: &$canLogin)
    }
}
```

```swift
class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Publishers.CombineLatest3($userName, $password, $passwordConfirmation)
            .map { userName, password, passwordConfirmation in
                return userName.isEmpty == false
                    && password.isEmpty == false
                    && password == passwordConfirmation
            }
            .assign(to: &$canLogin)
    }
}
```

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task {
            let combinedSequences = combineLatest(
                $userName.values,
                $password.values,
                $passwordConfirmation.values
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                canLogin = userName.isEmpty == false
                    && password.isEmpty == false
                    && password == passwordConfirmation
            }
        }
    }
}
```

```
@MainActor class LoginViewModel
```

The annotation `@MainActor` has been added to the declaration of `LoginViewModel`

It will make the compiler ensure that whenever we update the state of `LoginViewModel` it’s performed on the Main Thread 👌

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task {
            let combinedSequences = combineLatest(
                $userName.values,
                $password.values,
                $passwordConfirmation.values
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                canLogin = userName.isEmpty == false
                    && password.isEmpty == false
                    && password == passwordConfirmation
            }
        }
    }
}
```

```
@Published var userName: String = ""
@Published var password: String = ""
@Published var passwordConfirmation: String = ""

@Published private(set) var canLogin: Bool = false
```

Nothing has changed here! And that’s a pretty good thing!

It means we can migrate code from Combine to AsyncAlgorithm without changing the state SwiftUI will consume 👌

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task {
            let combinedSequences = combineLatest(
                $userName.values,
                $password.values,
                $passwordConfirmation.values
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                canLogin = userName.isEmpty == false
                    && password.isEmpty == false
                    && password == passwordConfirmation
            }
        }
    }
}
```

```
let combinedSequences = combineLatest(
    $userName.values,
    $password.values,
    $passwordConfirmation.values
)
```

`combineLatest()` is AsyncAlgorithm's equivalent of `Publishers.CombineLatest3()` in Combine

`$publisher.values` lets us turn a `Publisher` into an `AsyncPublisher` that conforms to `AsyncSequence`

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task {
            let combinedSequences = combineLatest(
                $userName.values,
                $password.values,
                $passwordConfirmation.values
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                canLogin = userName.isEmpty == false
                    && password.isEmpty == false
                    && password == passwordConfirmation
            }
        }
    }
}
```

```
for await (userName, password, passwordConfirmation) in combinedSequences {
    canLogin = userName.isEmpty == false
        && password.isEmpty == false
        && password == passwordConfirmation
}
```

Here we have the biggest difference with Combine!

We no longer use closure-based operators like `.map { }` to process values over time…

…and instead we use a new more imperative-styled syntax:

```
for await value in asyncSequence { }
```

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task {
            let combinedSequences = combineLatest(
                $userName.values,
                $password.values,
                $passwordConfirmation.values
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                canLogin = userName.isEmpty == false
                    && password.isEmpty == false
                    && password == passwordConfirmation
            }
        }
    }
}
```

```
Task {
    /* your async code */
}
```

Since our `for` loop will be continuously listening for new events, we need it to run into a `Task` of its own.

Otherwise, we would be blocking the main thread forever 🙃

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task {
            let combinedSequences = combineLatest(
                $userName.values,
                $password.values,
                $passwordConfirmation.values
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                canLogin = userName.isEmpty == false
                    && password.isEmpty == false
                    && password == passwordConfirmation
            }
        }
    }
}
```

```
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task {
            let combinedSequences = combineLatest(
                $userName.values,
                $password.values,
                $passwordConfirmation.values
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                canLogin = userName.isEmpty == false
                    && password.isEmpty == false
                    && password == passwordConfirmation
            }
        }
    }
}
```

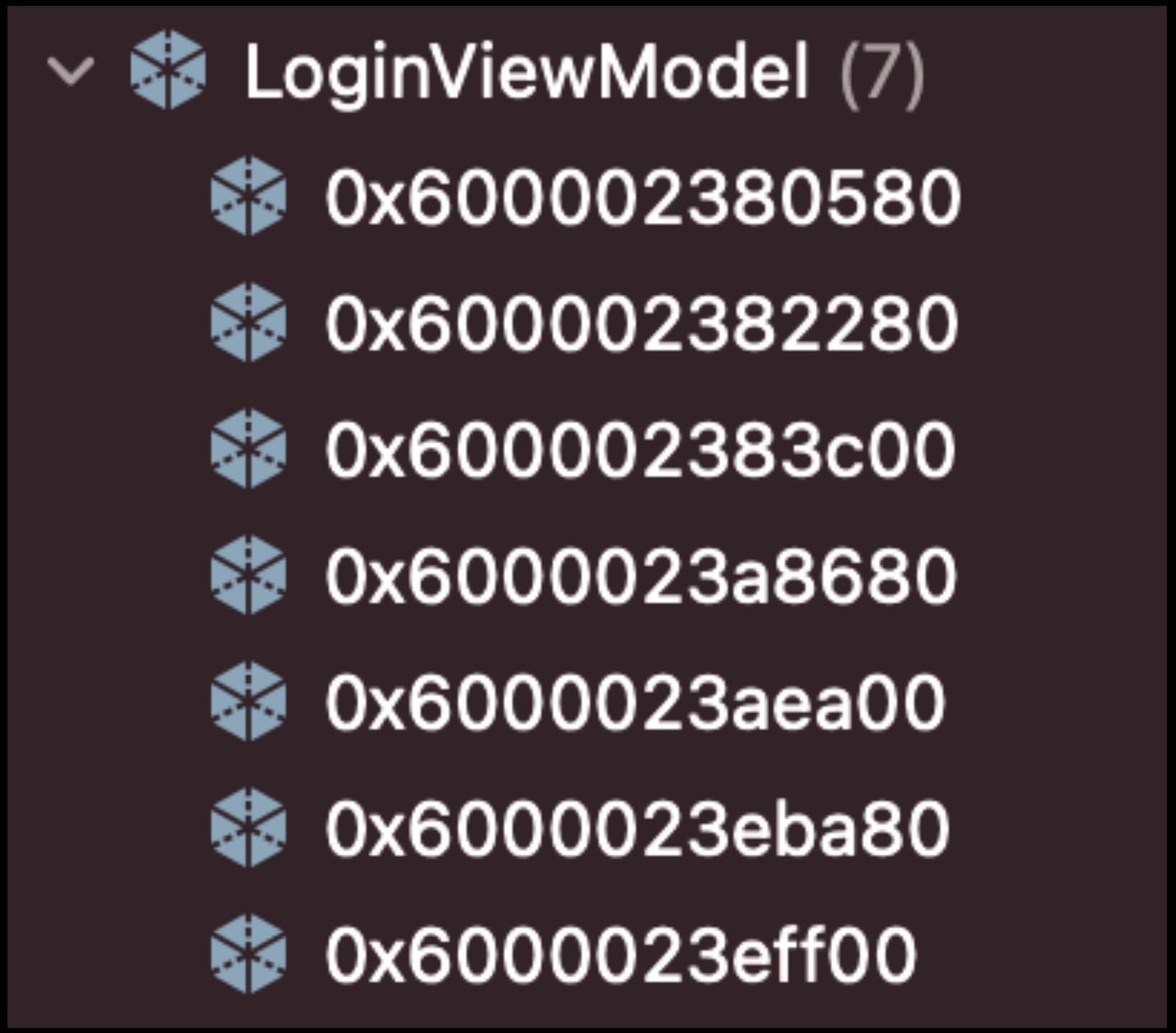
Good news: it’s pretty straightforward to go from Combine to AsyncAlgorithms 👌

```
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task {
            let combinedSequences = combineLatest(
                $userName.values,
                $password.values,
                $passwordConfirmation.values
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                canLogin = userName.isEmpty == false
                    && password.isEmpty == false
                    && password == passwordConfirmation
            }
        }
    }
}
```

Bad news: something is missing here! Can you spot what? 😐

```
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task {
            let combinedSequences = combineLatest(
                $userName.values,
                $password.values,
                $passwordConfirmation.values
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                canLogin = userName.isEmpty == false
                    && password.isEmpty == false
                    && password == passwordConfirmation
            }
        }
    }
}
```

There’s no code to cancel the `Task`, and the `Task` captures a strong reference to `LoginViewModel` 😱


LoginViewModel (7)
0x600002380580
0x600002382280
0x600002383c00
0x6000023a8680
0x6000023aea00
0x6000023eba80
0x6000023eff00

```
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task {
            let combinedSequences = combineLatest(
                $userName.values,
                $password.values,
                $passwordConfirmation.values
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                canLogin = userName.isEmpty == false
                    && password.isEmpty == false
                    && password == passwordConfirmation
            }
        }
    }
}
```

We’re gonna have to change our code a bit…

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task {
            let combinedSequences = combineLatest(
                $userName.values,
                $password.values,
                $passwordConfirmation.values
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                canLogin = userName.isEmpty == false
                    && password.isEmpty == false
                    && password == passwordConfirmation
            }
        }
    }
}
```

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task { [weak self] in
            guard let userNameSequence = self?.$userName.values,
                  let passwordSequence = self?.$password.values,
                  let passwordConfirmationSequence = self?.$passwordConfirmation.values
            else { return }

            let combinedSequences = combineLatest(
                userNameSequence,
                passwordSequence,
                passwordConfirmationSequence
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                self?.canLogin = userName.isEmpty == false
                && password.isEmpty == false
                && password == passwordConfirmation
            }
        }
    }
}
```

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task { [weak self] in
            guard let userNameSequence = self?.$userName.values,
                  let passwordSequence = self?.$password.values,
                  let passwordConfirmationSequence = self?.$passwordConfirmation.values
            else { return }

            let combinedSequences = combineLatest(
                userNameSequence,
                passwordSequence,
                passwordConfirmationSequence
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                self?.canLogin = userName.isEmpty == false
                && password.isEmpty == false
                && password == passwordConfirmation
            }
        }
    }
}
```

No more leaking `LoginViewModel` 👌

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task { [weak self] in
            guard let userNameSequence = self?.$userName.values,
                  let passwordSequence = self?.$password.values,
                  let passwordConfirmationSequence = self?.$passwordConfirmation.values
            else { return }

            let combinedSequences = combineLatest(
                userNameSequence,
                passwordSequence,
                passwordConfirmationSequence
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                self?.canLogin = userName.isEmpty == false
                && password.isEmpty == false
                && password == passwordConfirmation
            }
        }
    }
}
```

But a good amount of boilerplate code 😔

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        Task { [weak self] in
            guard let userNameSequence = self?.$userName.values,
                  let passwordSequence = self?.$password.values,
                  let passwordConfirmationSequence = self?.$passwordConfirmation.values
            else { return }

            let combinedSequences = combineLatest(
                userNameSequence,
                passwordSequence,
                passwordConfirmationSequence
            )

            for await (userName, password, passwordConfirmation) in combinedSequences {
                self?.canLogin = userName.isEmpty == false
                && password.isEmpty == false
                && password == passwordConfirmation
            }
        }
    }
}
```

Maybe we can do something about it!

```
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        let combinedSequences = combineLatest(
            $userName.values,
            $password.values,
            $passwordConfirmation.values
        )

        Task { [weak self] in
            for await (userName, password, passwordConfirmation) in combinedSequences {
                self?.canLogin = userName.isEmpty == false
                && password.isEmpty == false
                && password == passwordConfirmation
            }
        }
    }
}
```

Much better 👌

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        let combinedSequences = combineLatest(
            $userName.values,
            $password.values,
            $passwordConfirmation.values
        )

        Task { [weak self] in
            for await (userName, password, passwordConfirmation) in combinedSequences {
                self?.canLogin = userName.isEmpty == false
                && password.isEmpty == false
                && password == passwordConfirmation
            }
        }
    }
}
```

You might be thinking: why create a `Task` and not just use the modifier `.task { }`? 🤨

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        let combinedSequences = combineLatest(
            $userName.values,
            $password.values,
            $passwordConfirmation.values
        )

        Task { [weak self] in
            for await (userName, password, passwordConfirmation) in combinedSequences {
                self?.canLogin = userName.isEmpty == false
                && password.isEmpty == false
                && password == passwordConfirmation
            }
        }
    }
}
```

You could use the modifier `.task { }`, but it would:

1. Make the `ViewModel` rely on the `View` initializing it properly…
2. Require a call to `.task { }` for each event loop…

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        let combinedSequences = combineLatest(
            $userName.values,
            $password.values,
            $passwordConfirmation.values
        )

        Task { [weak self] in
            for await (userName, password, passwordConfirmation) in combinedSequences {
                self?.canLogin = userName.isEmpty == false
                && password.isEmpty == false
                && password == passwordConfirmation
            }
        }
    }
}
```

Are we sure the `Task` eventually cancels? 🤔

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        let combinedSequences = combineLatest(
            $userName.values,
            $password.values,
            $passwordConfirmation.values
        )

        Task { [weak self] in
            for await (userName, password, passwordConfirmation) in combinedSequences {
                self?.canLogin = userName.isEmpty == false
                && password.isEmpty == false
                && password == passwordConfirmation
            }
        }
    }
}
```

Here it should cancel, but what about the general case? 😐

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        let combinedSequences = combineLatest(
            $userName.values,
            $password.values,
            $passwordConfirmation.values
        )

        Task { [weak self] in
            for await (userName, password, passwordConfirmation) in combinedSequences {
                guard let self else { return }

                self.canLogin = userName.isEmpty == false
                && password.isEmpty == false
                && password == passwordConfirmation
            }
        }
    }
}
```

```
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        let combinedSequences = combineLatest(
            $userName.values,
            $password.values,
            $passwordConfirmation.values
        )

        Task { [weak self] in
            for await (userName, password, passwordConfirmation) in combinedSequences {
                guard let self else { return }

                self.canLogin = userName.isEmpty == false
                && password.isEmpty == false
                && password == passwordConfirmation
            }
        }
    }
}
```

Notice how the strong **`self`** is local to the **`for`** loop!

```swift
@MainActor class LoginViewModel: ObservableObject {

    @Published var userName: String = ""
    @Published var password: String = ""
    @Published var passwordConfirmation: String = ""

    @Published private(set) var canLogin: Bool = false

    init() {
        let combinedSequences = combineLatest(
            $userName.values,
            $password.values,
            $passwordConfirmation.values
        )

        Task { [weak self] in
            for await (userName, password, passwordConfirmation) in combinedSequences {
                guard let self else { return }

                self.canLogin = userName.isEmpty == false
                && password.isEmpty == false
                && password == passwordConfirmation
            }
        }
    }
}
```

Alternatively, you could also ensure `Task` cancellation using `AnyCancellable` 🙃

AsyncAlgorithms also features interesting abstractions!

AsyncChannel<T>

```
let channel = AsyncChannel<String>()
```

```
let channel = AsyncChannel<String>()

// Producer
Task {
    while let resultOfLongCalculation = doLongStuff() {
        await channel.send(resultOfLongCalculation)
    }
    channel.finish()
}
```

```
let channel = AsyncChannel<String>()

// Producer
Task {
    while let resultOfLongCalculation = doLongStuff() {
        await channel.send(resultOfLongCalculation)
    }
    channel.finish()
}

// Consumer
for await calculationResult in channel {
    print(calculationResult)
}
```

It’s a nice API to implementing a waiting queue on a ViewModel!

```swift
@MainActor class MoviesViewModel: ObservableObject {

    enum Command {
        case fetchInitialData
        case fetchMoreData
        case favoriteMovie(Movie)
    }

    let commandChannel = AsyncChannel<Command>()

    /* ... */

    init() {
        Task { [weak self, commandChannel] in
            for await command in commandChannel {
                switch command {
                case .fetchInitialData:
                    await self?.fetchInitialData()
                case .fetchMoreData:
                    await self?.fetchMoreData()
                case .favoriteMovie(let movie):
                    await self?.favorite(movie: movie)
                }
            }
        }
    }

    private func fetchInitialData() async {
        currentPage = 1
        upcomingMovies = await getMovies().results
    }

    private func fetchMoreData() async {
        currentPage += 1
        upcomingMovies += await getMovies(page: currentPage).results
    }

    private func favorite(movie: Movie) async {
        /* ... */
    }
}
```

```
@MainActor class MoviesViewModel: ObservableObject {

    enum Command {
        case fetchInitialData
        case fetchMoreData
        case favoriteMovie(Movie)
    }

    let commandChannel = AsyncChannel<Command>()

    /* ... */
}
```

We define a set of `Command` that `MoviesViewModel` can handle…

…and it uses an `AsyncChannel<Command>` to receive them

```swift
@MainActor class MoviesViewModel: ObservableObject {

    enum Command {
        case fetchInitialData
        case fetchMoreData
        case favoriteMovie(Movie)
    }

    let commandChannel = AsyncChannel<Command>()

    /* ... */

    init() {
        Task { [weak self, commandChannel] in
            for await command in commandChannel {
                switch command {
                case .fetchInitialData:
                    await self?.fetchInitialData()
                case .fetchMoreData:
                    await self?.fetchMoreData()
                case .favoriteMovie(let movie):
                    await self?.favorite(movie: movie)
                }
            }
        }
    }

    private func fetchInitialData() async {
        currentPage = 1
        upcomingMovies = await getMovies().results
    }

    private func fetchMoreData() async {
        currentPage += 1
        upcomingMovies += await getMovies(page: currentPage).results
    }

    private func favorite(movie: Movie) async {
        /* ... */
    }
}
```

```
init() {
    Task { [weak self, commandChannel] in
        for await command in commandChannel {
            switch command {
            case .fetchInitialData:
                await self?.fetchInitialData()
            case .fetchMoreData:
                await self?.fetchMoreData()
            case .favoriteMovie(let movie):
                await self?.favorite(movie: movie)
            }
        }
    }
}
```

Thanks to the `AsyncChannel` we can listen to the commands…

…and execute them one after the other 👌

```swift
@MainActor class MoviesViewModel: ObservableObject {

    enum Command {
        case fetchInitialData
        case fetchMoreData
        case favoriteMovie(Movie)
    }

    let commandChannel = AsyncChannel<Command>()

    /* ... */

    init() {
        Task { [weak self, commandChannel] in
            for await command in commandChannel {
                switch command {
                case .fetchInitialData:
                    await self?.fetchInitialData()
                case .fetchMoreData:
                    await self?.fetchMoreData()
                case .favoriteMovie(let movie):
                    await self?.favorite(movie: movie)
                }
            }
        }
    }

    private func fetchInitialData() async {
        currentPage = 1
        upcomingMovies = await getMovies().results
    }

    private func fetchMoreData() async {
        currentPage += 1
        upcomingMovies += await getMovies(page: currentPage).results
    }

    private func favorite(movie: Movie) async {
        /* ... */
    }
}
```

```swift
private func fetchInitialData() async {
    currentPage = 1
    upcomingMovies = await getMovies().results
}

private func fetchMoreData() async {
    currentPage += 1
    upcomingMovies += await getMovies(page: currentPage).results
}

private func favorite(movie: Movie) async {
    /* ... */
}
```

Finally, the methods that implement the commands are just regular **`async`** methods 👌

```swift
@MainActor class MoviesViewModel: ObservableObject {

    enum Command {
        case fetchInitialData
        case fetchMoreData
        case favoriteMovie(Movie)
    }

    let commandChannel = AsyncChannel<Command>()

    /* ... */

    init() {
        Task { [weak self] in
            for await command in commandChannel {
                switch command {
                case .fetchInitialData:
                    await self?.fetchInitialData()
                case .fetchMoreData:
                    await self?.fetchMoreData()
                case .favoriteMovie(let movie):
                    await self?.favorite(movie: movie)
                }
            }
        }
    }

    private func fetchInitialData() async {
        currentPage = 1
        upcomingMovies = await getMovies().results
    }

    private func fetchMoreData() async {
        currentPage += 1
        upcomingMovies += await getMovies(page: currentPage).results
    }

    private func favorite(movie: Movie) async {
        /* ... */
    }
}
```

Now let’s have a look at the View

```swift
struct MoviesView: View {

    @StateObject var viewModel = MoviesViewModel()

    var body: some View {
        List(viewModel.movies) { movie in
            NavigationLink {
                MovieDetailsView(movie: movie)
            } label: {
                HStack {
                    AsyncImage(url: movie.posterURL) { poster in
                        poster
                            .resizable()
                            .aspectRatio(contentMode: .fill)
                            .frame(width: 100)
                    } placeholder: {
                        ProgressView()
                            .frame(width: 100)
                    }

                    VStack(alignment: .leading) {
                        Text(movie.title)
                            .font(.headline)
                        Text(movie.overview)
                            .font(.caption)
                            .lineLimit(3)
                    }
                }
            }
            .task {
                if movie == viewModel.movies.last {
                    await viewModel.commandChannel.send(.fetchMoreData)
                }
            }
        }
        .navigationTitle("Upcoming Movies")
        .task {
            await viewModel.commandChannel.send(.fetchInitialData)
        }
       .refreshable {
            await viewModel.commandChannel.send(.fetchInitialData)
        }
    }
}
```

```
var body: some View {
        List(viewModel.movies) { movie in
            NavigationLink {
                MovieDetailsView(movie: movie)
            } label: {
                HStack {
                    /* ... */
                }
            }
            .task {
                if movie == viewModel.movies.last {
                    await viewModel.commandChannel.send(.fetchMoreData)
                }
            }
        }
        .navigationTitle("Upcoming Movies")
        .task {
            await viewModel.commandChannel.send(.fetchInitialData)
        }
       .refreshable {
            await viewModel.commandChannel.send(.fetchInitialData)
        }
    }
}
```

```
var body: some View {
        List(viewModel.movies) { movie in
            NavigationLink {
                MovieDetailsView(movie: movie)
            } label: {
                HStack {
                    /* ... */
                }
            }
            .task {
                if movie == viewModel.movies.last {
                    await viewModel.commandChannel.send(.fetchMoreData)
                }
            }
        }
        .navigationTitle("Upcoming Movies")
        .task {
            await viewModel.commandChannel.send(.fetchInitialData)
        }
       .refreshable {
            await viewModel.commandChannel.send(.fetchInitialData)
        }
    }
}
```

```
var body: some View {
        List(viewModel.movies) { movie in
            NavigationLink {
                MovieDetailsView(movie: movie)
            } label: {
                HStack {
                    /* ... */
                }
            }
            .task {
                if movie == viewModel.movies.last {
                    await viewModel.commandChannel.send(.fetchMoreData)
                }
            }
        }
        .navigationTitle("Upcoming Movies")
        .task {
            await viewModel.commandChannel.send(.fetchInitialData)
        }
       .refreshable {
            await viewModel.commandChannel.send(.fetchInitialData)
        }
    }
}
```

`MoviesView` can still send commands concurrently, but `MoviesViewModel` model will handle them sequentially 👌

Which is better between Combine and AsyncAlgorithms?

If you really dislike the functional reactive syntax, AsyncAlgorithms offers a nice alternative

However, AsyncAlgorithms also comes with pitfalls of its own and is less battle-tested

In 2022, having a project that relies entirely on AsyncAlgorithms is a bit of a risky move…

…but that might change after WWDC 2023: wait & see!


That's all Folks!

Thank You! 

Twitter

YouTube

https://www.photoroom.com/company/