AsyncAlgorithms: an alternative to Combine?

Vincent Pradeilles (@v_pradeilles) – PhotoRoom

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
```

```
@MainActor class LoginViewModel
```

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

```
@MainActor class LoginViewModel
```

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

```
@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 is performed on the Main Thread 👌

```
@MainActor class 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 {
            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
```

```
@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!

```
@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!

```
@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 👌

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

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

```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
)
```

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

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

```
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`

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

```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
}
```

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

Here we have the biggest difference with Combine!

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

Here we have the biggest difference with Combine!

```
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…

```
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…

```
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 (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 { }
```

```
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 {
            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 */
}
```

```
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.

```
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.

```
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 🙃

```
Task {
    /* your async 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 {
            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 {
            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 👌

```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 {
            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
            }
        }
    }
}
```

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

```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 {
            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
            }
        }
    }
}
```

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

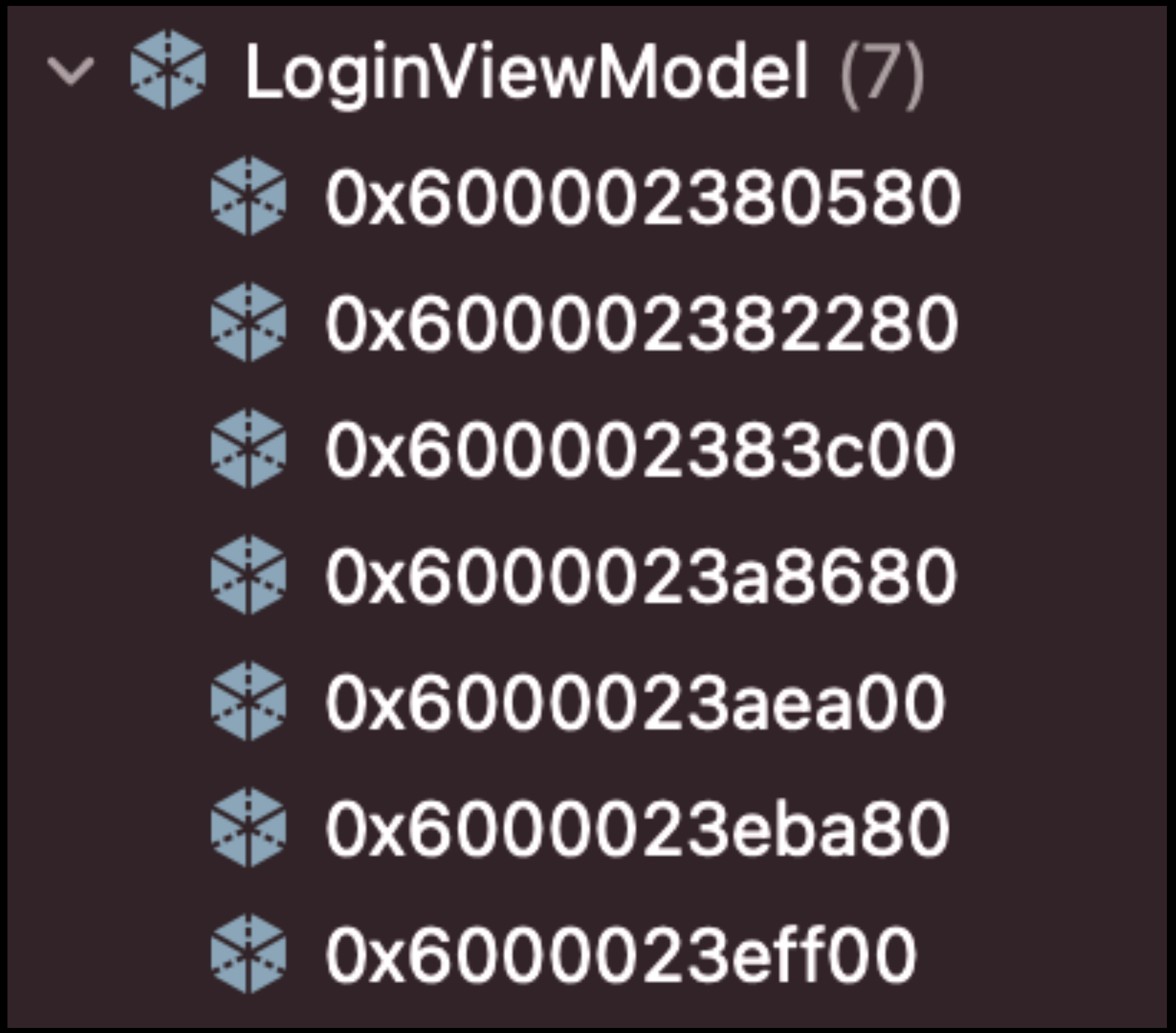
```swift
@MainActor class LoginViewModel: ObservableObject {

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

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

    init() {
        Task {
            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
            }
        }
    }
}
```


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

```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
            }
        }
    }
}
```

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 {
            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
            }
        }
    }
}
```

```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
            }
        }
    }
}
```

```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
            }
        }
    }
}
```

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
            }
        }
    }
}
```

```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
            }
        }
    }
}
```

Maybe we can do something about it!

```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() {
        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
            }
        }
    }
}
```

```
@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
            }
        }
    }
}
```

```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
            }
        }
    }
}
```

```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
            }
        }
    }
}
```

```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
            }
        }
    }
}
```

```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:

```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:

```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…

```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
            }
        }
    }
}
```

```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
            }
        }
    }
}
```

```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
            }
        }
    }
}
```

```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
            }
        }
    }
}
```

```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 {
                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() {
        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
            }
        }
    }
}
```

```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
            }
        }
    }
}
```

```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
            }
        }
    }
}
```

```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` 🙃

```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
            }
        }
    }
}
```

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 implement 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>()

    /* ... */
}
```

```
@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…

```
@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…

```
@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

```
@MainActor class MoviesViewModel: ObservableObject {

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

    let commandChannel = AsyncChannel<Command>()

    /* ... */
}
```

```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)
            }
        }
    }
}
```

```
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…

```
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…

```
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 👌

```
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)
            }
        }
    }
}
```

```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 {
    /* ... */
}
```

```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
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
@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)
        }
    }
}
```

```swift
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 👌

```swift
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)
        }
    }
}
```

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 early 2023, having a project that relies entirely on AsyncAlgorithms is a bit of a risky move…

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


That's all Folks!

Thank You! 

Thank you! 😊


And what is the motto
of House AppStore Connect?
“We noticed an issue
with your submission”
…a common saying, yes.
But not their official motto.
imgflip.com

Twitter

YouTube

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