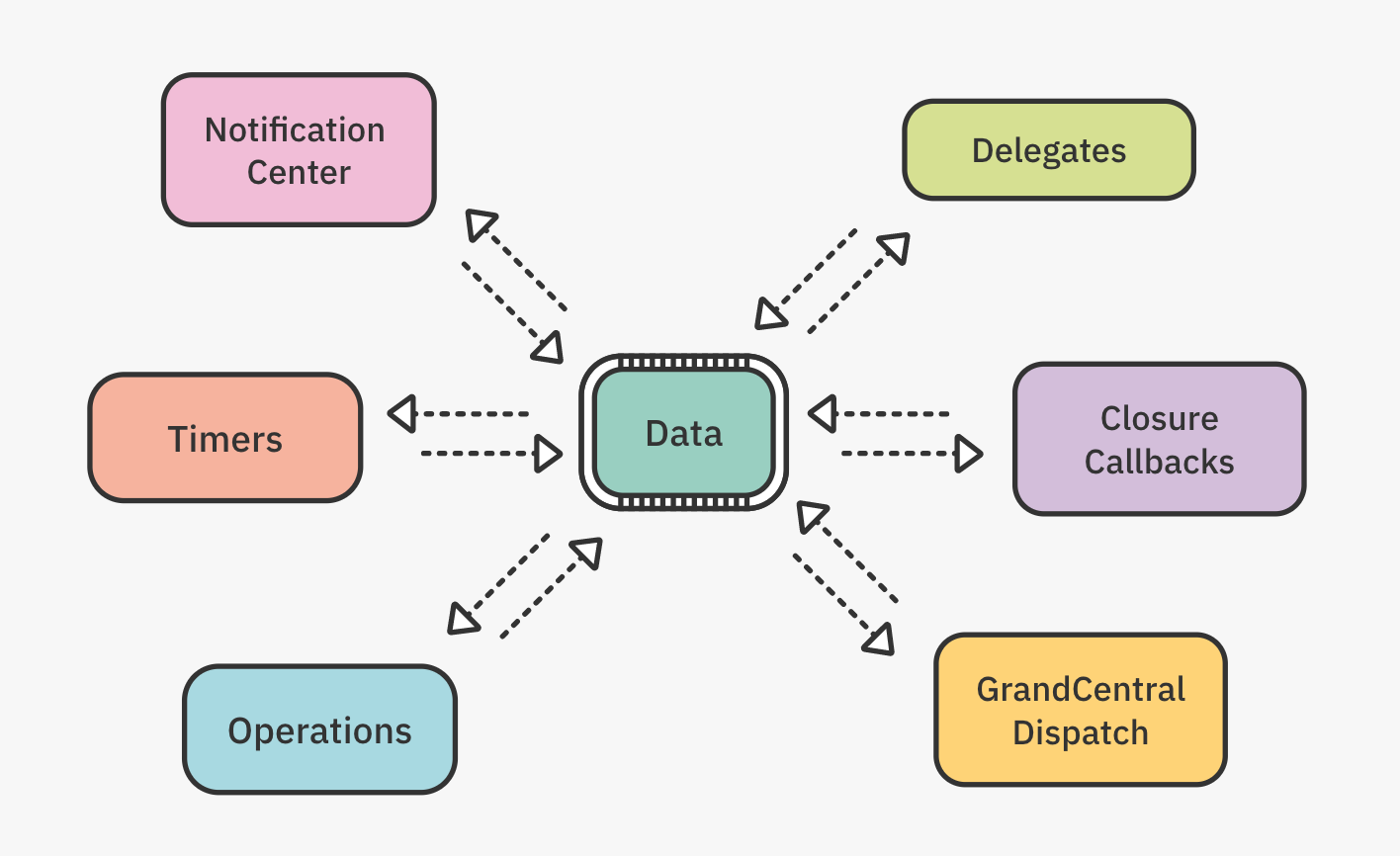
# Combine



<https://fxstudio.dev/combine-vs-mvvm-overview/>

1. **What is Combine:**

<https://fxstudio.dev/hello-combine/>

<https://www.kodeco.com/books/combine-asynchronous-programming-with-swift/v3.0/chapters/6-time-manipulation-operators>

- Reactive Programing: is programing with asynchronous data streams.

- Data stream: has 3 main types: value, error and complete

An example about asynchronous program:

Bạn có 1 button, mỗi lần nhấn vào nó thì bạn sẽ đếm 1 lần.

Bạn không biết được thời điểm nào sẽ bấm? bấm bao nhiêu lần? và lúc nào kết thúc?

Đó chính là mô tả đơn giản nhất về luồng dữ liệu bất đồng bộ. Khi giải quyết chúng, thì ta sẽ *không quan tâm tới các đối tượng*. *Chỉ quan tâm tới dữ liệu vào ra* và *phản ứng lại với từng giá trị* trên stream.

-Function reactive Programing (FRP): will use stream to transfer datas as value, error, complete.

- FRP: cho phép filter (filter, take, scan, …), chuyển đổi từ stream này qua stream khác (map, flatMap, reduce), hoặc merge nhiều stream thành một stream mới (combine, merge, zip, …) khá dễ dàng mà không làm thay đổi trạng thái của stream ban đầu.

1. **Combine frameworks of Apple:**

- problem is multi-thread.

- các ứng dụng thì luôn tồn tại 1 thread là Main Thread để duy trì giao diện ứng dụng. Các tương tác xử lý như API, Location, Upload … để phải thực thi ở 1 Thread khác. Hoặc việc người dùng luôn tác động giao diện và ứng dụng ta phải phản ứng lại các tác động của người dùng … Tất cả chúng đều buộc chúng ta phải xử lý bất đồng bộ với nhau.

* 1. Foundation and UIKIT/AppKit:

Apple created method to perform asynchronous:

- Notification Center: Thực thi 1 đoạn mã bất cứ khi nào sự kiện mà bạn quan tâm xảy ra.

- Delegate: Cho phép bạn xác định 1 đối tượng hoạt động thay mặt hoặc phối hợp với một đối tượng khác.

- Grand cental Dispatch and Operation Queue: Trừu tượng hoá trong việc thực hiện từng phần của công việc, Lên lịch thực hiện từng phần, theo nhiều cách khác nhau trong ngăn xếp (tuần tự hay ưu tiên).

- Closures: Tạo ra các đoạn mã có thể tách rời và chuyển cho các đối tượng khác thực thi ở một nơi nào đó.

- **RxSwift**: là người tiền nhiệm của Combine, is third-party.

Rx = OBSERVABLE + OBSERVER + SCHEDULERS

- **Combine**: is framework of apple. Định nghĩa một chuẩn riêng, Vẫn giống như Rx về cú pháp, bản chất và cơ chế.

-**SwiftUI**: xem như là một ngôn ngữ mới. Mục đích duy nhất của nó là bạn chỉ cần code 1 lần và chạy trên tất cả mọi nền tảng.

- **Catalyst**: nỗ lực đưa ứng dụng trên iPadOS lên trên MacOS.

1. **Components of Combine:**

- There are 3 main components: Publisher, operator and subcribers.

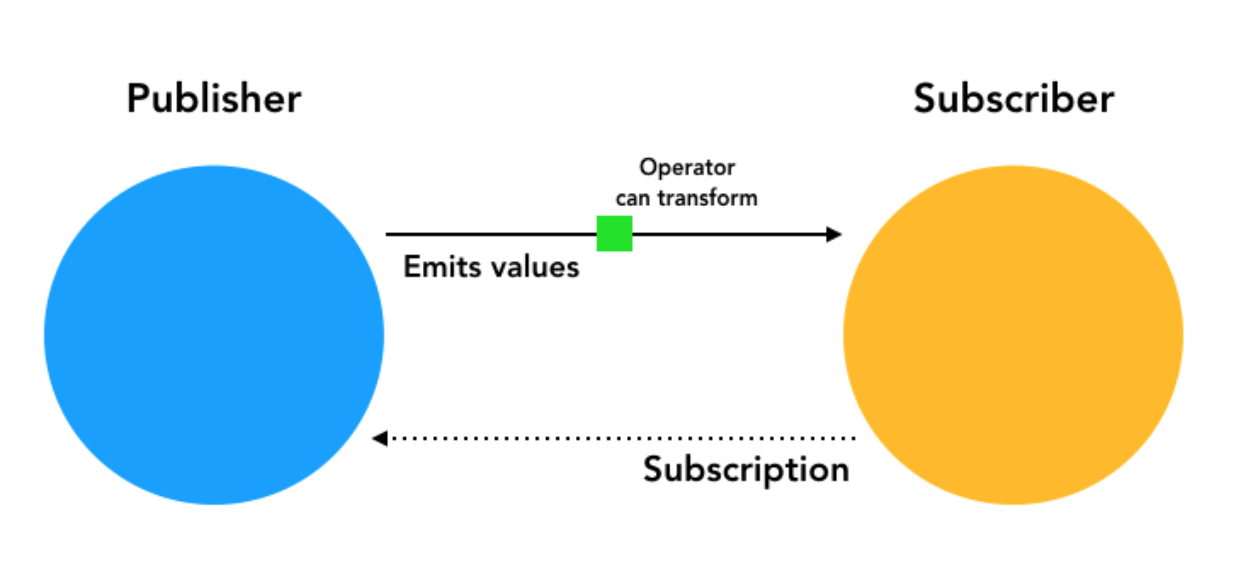
- Publishers are types that can emit values over time to one or more interested parties, such as subscribers.

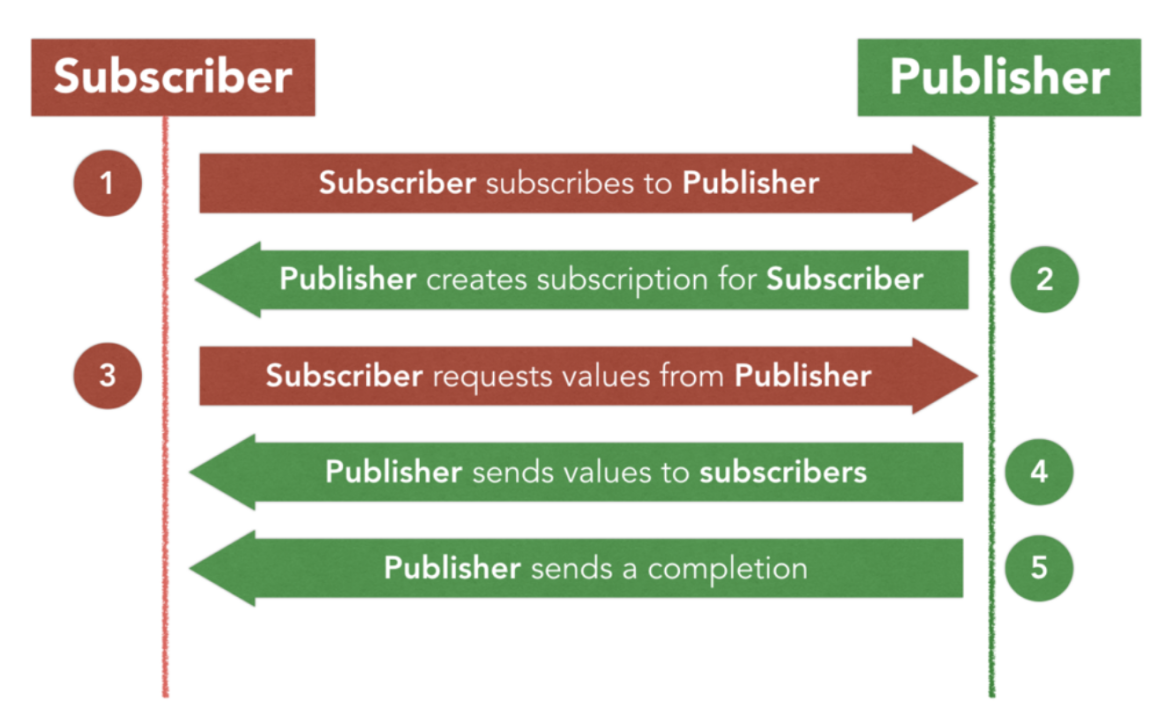
- Các subscriber không nhất thiết phải ở gần, hay cùng chung class với các publisher. Hay cùng chung 1 queue hoặc 1 thread. Mà chúng ta có thể phát và nhận ở bất kì đâu.

- Subcriber must register with Publisher, then Publisher will emit data, bằng sink hoặc assign.

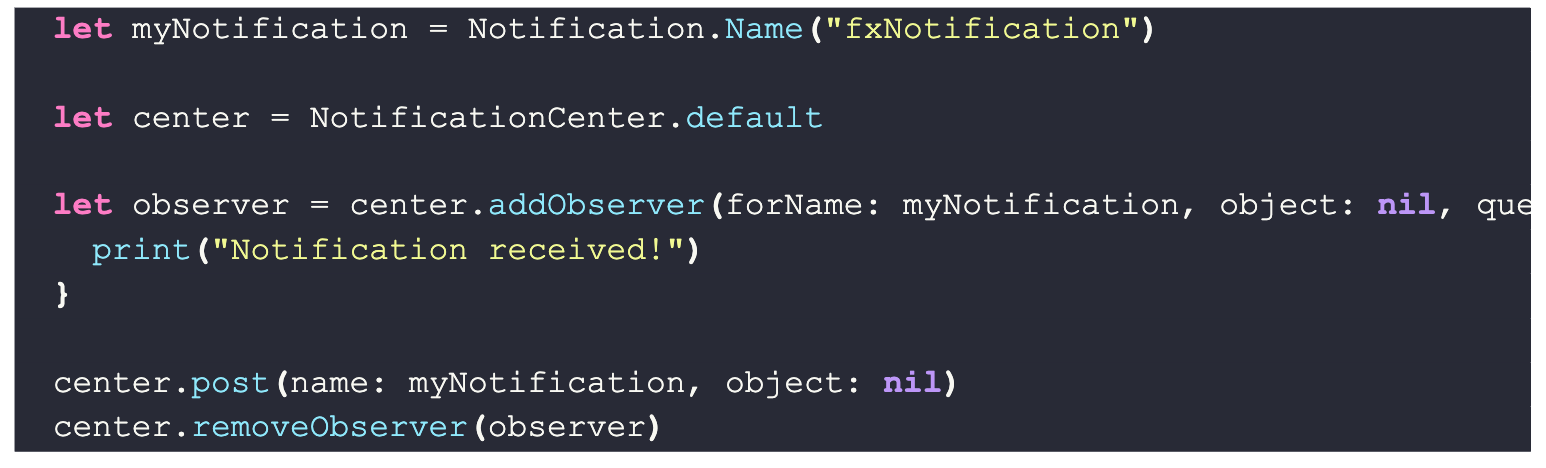
- Operator: là các phương thức để biến đổi data.

- Subcription: la việc Subscriber đăng ký tới Publisher.



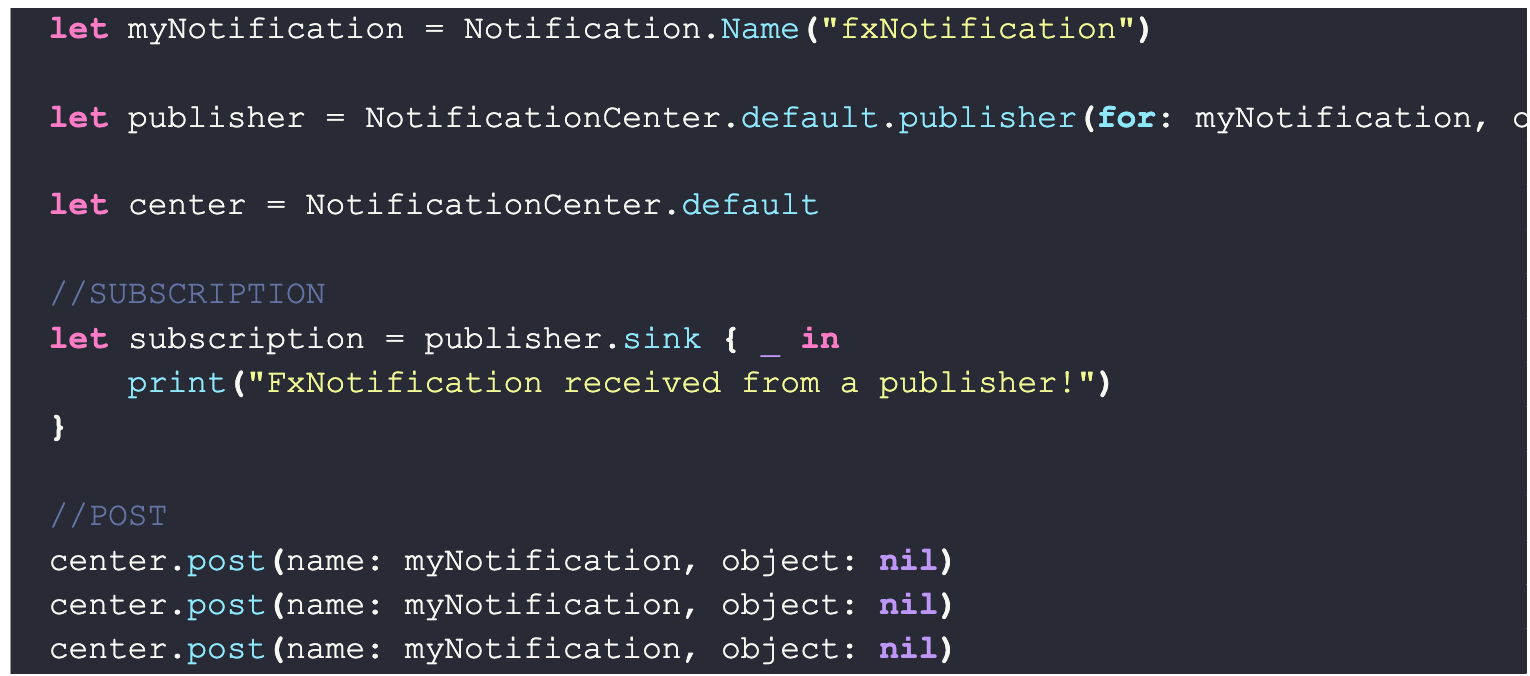


- Notification: cũng có chứa 1 phần của Combine. Phải khai báo notification, add observer cho notification để lắng nghe, khi muốn phát noti thì fai gọi fucntion Post, remove noti khi khong dung den nua.



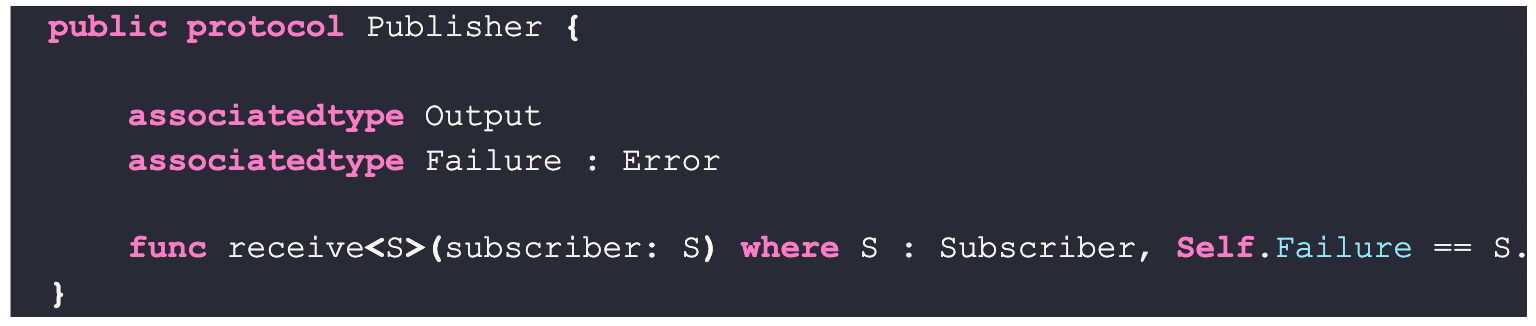
- Notification khi dung combine framework:

Bạn không cần quan tâm tới việc phải add và remove các observer.



* 1. **Publisher:**

Publisher chính là trái tim của toàn bộ Combine. Chịu trách nhiệm phát đi các giá trị và quản lý các subscription tới nó. Nó chính là nguồn phát.



Tất cả các Publisher đều phải kế thừa protocol này.

Nếu publisher phát ra complete hay error thì sẽ không phát thêm được nữa. Nó sẽ kết thúc ở đây.

- Output : chính là kiểu giá trị cho dữ liệu bạn phát ra

- Failure: kiểu dữ liệu cho trường hợp lỗi.

- value : chính là cái chúng ta cần, dữ liệu

- error : nếu gặp lỗi thì các subscriber sẽ nhận được như vậy

- complete : là kết thúc vòng đời đau khổ của 1 publisher

**Các loại Publisher:**

**<https://fxstudio.dev/ho-hang-nha-publisher-trong-combine/>**

- ***Publisher từ property của class***: @Published **var** name: String

Thêm toán tử dấu $ để truy cập tới nó

+ phải khởi tạo giá trị ban đầu lúc khai báo.

+ Không ảnh hưởng gì tới code của class chứa nó, chỉ khai báo thêm từ khoá @Publisher phía trước.

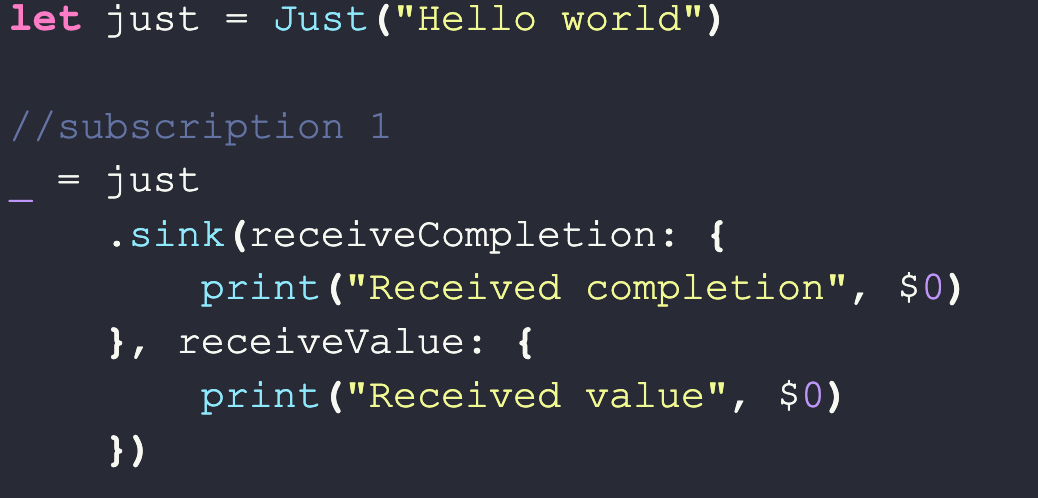
+ Với Output là cùng kiểu dữ liệu với property đó. Và không bao giờ có lỗi.

+ Vừa lưu trữ được giá trị và phát đi được giá trị.

+ Real-time, bất cứ khi nào bạn thay đổi giá trị thì đồng thời nó sẽ phát đi giá trị đó cho các subscriber.

+ private hay public đều được.

- ***Just***: sẽ phát ra 1 giá trị duy nhất tới subscriber và sau đó là finished. Phải cung cấp giá trị ban đầu khi init Just.



- ***Future***:

+ Là một Class

+ Là một Publisher

+ Đối tượng này sẽ phát ra một giá trị duy nhất, sau đó kết thúc hoặc fail.

+ Nó sẽ thực hiện một lời hứa Promise. Đó là 1 closure với kiểu Result, nên sẽ có 1 trong 2 trường hợp: success, failure.

+ Khi hoạt động: Lần subscribe đầu tiên, nó sẽ thực hiện đầy đủ các thủ tục. Và phát ra giá trị, sau đó kết thúc hoặc thất bại. Lần subscribe tiếp theo, chỉ phát ra giá trị cuối cùng. Bỏ qua các bước thủ thục khác.

**- *Subject*:**

+ là nó cũng là 1 loại Publisher.

+ Là thực thể kết nối giữa code Combine và Non-Combine.

**+ *PassthroughSubject***: lúc nào phát thì sẽ nhận được giá trị. Ưu điểm là có cái gì là ném cái đó đi. Chứ không quan tâm phải gởi dữ liệu ngay từ đầu cho subscriber. Hiệu quả trong việc gởi các sự kiện hoặc call back. Không cần cung cấp giá trị ban đầu cho việc khai báo subject này.

**+ *CurrentValueSubject*** : không quan tâm lúc nào phát, chỉ cần subscription là có giá trị (cuối cùng). Ưu điểm của nó khi có subscription tới thì subscriber sẽ có ngay dữ liệu liền. Hiệu quả khi lưu trữ dữ liệu. Cần cung cấp giá trị của Output ban đầu khi khai báo subject này.

**- *Type Erasure*:**

+ được dùng khi bạn muốn subscribe tới publisher mà không cần biết quá nhiều về kiểu dữ liệu của publisher.

+ Type-erased publisher gồm các class: AnyPublisher, AnyCancellable (nó giúp cho quá trình tự huỷ của subscription xảy ra).

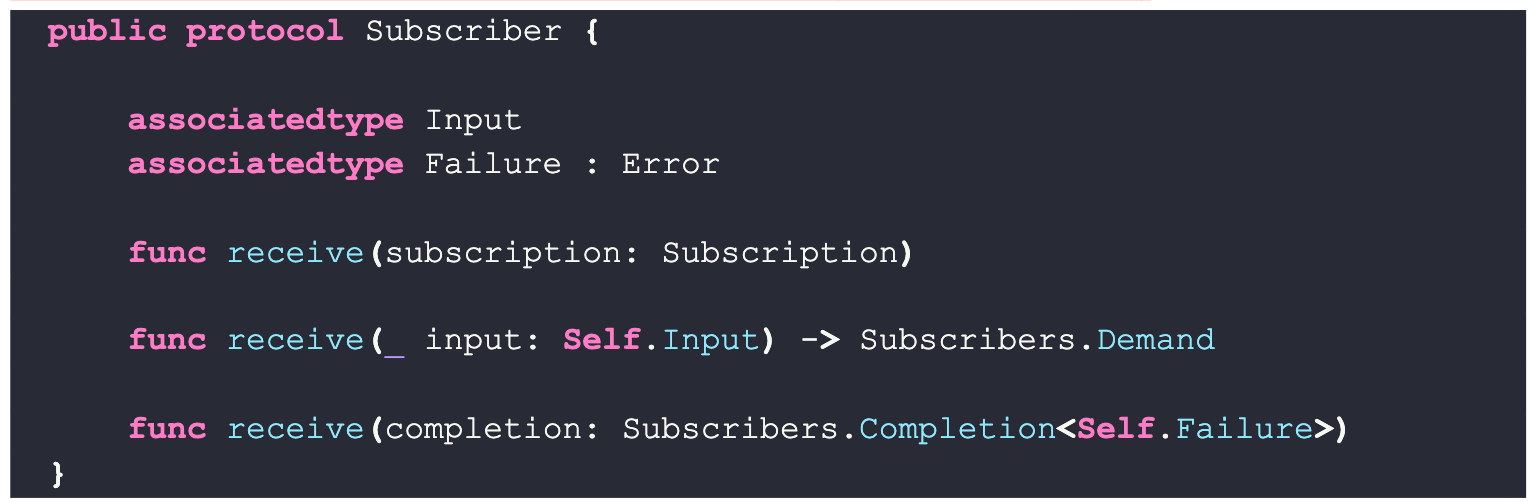
+ Với AnyPublisher, thì không thể gọi function send(\_:) được.

+ Class này đã bọc và ẩn đi nhiều phương thức & thuộc tính của Publisher.

+ Trong thực tế, bạn cũng không nên lạm dụng hay khuyến khích dùng nhiều kiểu này. Vì đôi khi bạn cần khai báo và xác định rõ kiểu giá trị nhận được.

* 1. **Subcriber:**

Nó sẽ nhận các giá trị từ Publisher. Và cũng như publisher, thì tất cả các subscriber để phải kế thừa protocol Subscriber.



- receive(subscription:) khi nhận được subscription từ Publisher. Lúc này Subscriber vẫn có quyền quyết định việc lấy bao nhiêu dữ liệu từ Publisher.

- receive(input:) khi nhận được giá trị từ Publisher và chúng ta sẽ điểu chỉnh request tiếp dữ liệu thông qua Demand . Có nghĩa bạn muốn nhận tiếp hay không hay nhận hết, thì bạn có thể tuỳ ý quyết định … đây là ưu điểm mà Combine hơn người tiền nhiệm RxSwift.

- receive(completion:) khi nhận completion từ publisher.

- Subscriber hỗ trợ việc tự huỷ khi subscription ngắt kết nối. có 2 kiểu huỷ:

+ Tự động huỷ thông AnyCancellable, đó là việc tạo ra các subscriber bằng sink hoặc assign.

+ Huỷ bằng tay với việc subscriber gọi hàm cancel() của nó.

**Cac cach tao Subcriber:**

**- Sink:**

Là cách đơn giản nhất mà bạn có thể dùng để tạo ra 1 Subscriber.

Bạn cung cấp cho nó 1 closure để xử lý các giá trị nhận được hay completion từ Publisher.

+ ta có thể xử lí luôn việc có error phát sinh.

**- Assign**

**Screen Shot 2023-05-30 at 11.06.54**

+ với Subscribers.Assign thì ta có được 1 Subscriber và nó cũng là Cancellable.

+ Ưu điểm: giúp binding dữ liệu lên đối tượng một cách nhanh chóng.

+ Nhược điểm: publisher phải đảm bảo là không bao giờ phát đi error

+ The assign(to:) operator doesn’t return an AnyCancellable token, because it manages the lifecycle internally and cancels the subscription when the @Published property deinitializes.

**- Cancelable:**

+ là 1 type-erasing class, tạo ra 1 đối tượng sẽ tự động huỷ. Khi nó huỷ thì các subscription sẽ bị huỷ theo. Và các Subscriber có implement nó cũng tự động huỷ theo.

+ Nếu như bạn không kiên nhẫn để chờ huỹ, thì có thể sử dụng function cancel() của subscription. Mọi việc sẽ được giải quyết.

+ sink và assign sẽ tạo ra đối tượng AnyCancellable.

+ không thể tạo ra nhiều đối tượng cancellable cho một lần subscribe. Nên cần phải quản lý tập trung: dùng Set<AnyCancellable>()

1. **Operators:**

- là các từ vựng, giúp bạn diễn đạt logic/ý nghĩa của mình trong code.

1. **Operators in Combine:**

<https://fxstudio.dev/combine-transforming-operators-trong-10-phut/>

* 1. **Transforming operators**:

- Publisher:

- collect(): way to transform a stream of individual values from a publisher into a single array.

collect(10)

- map()

- tryMap(): likes map(), but it can catch error to throw

- flatMap(maxPublishers:\_:)

flatMap(maxPublishers: .max(2)) { $0.value }

+ biến đổi 1 publisher này thành 1 publisher khác, Mới hoàn toàn.

+ Thường sử dụng flatMap để truy cập vào các thuộc tính trong của 1 publisher.

+ giúp cho việc hợp nhất các stream của các publisher thành 1 stream và đại diện chung là 1 publisher mới với kiểu khác các publisher kia. quản lí các stream của các publisher trong đó. Hiểu nôm na là hợp nhất các stream thành 1 steam và khống chế số lượng các steam lắng nghe: maxPublishers

- Scan:

scan(0) { $0 + $1 }

- replaceNil():

replaceNil(with: 2)

- replaceEmpty(with:)

replaceEmpty(with: 1)

* 1. **Filtering operators**:

<https://fxstudio.dev/combine-filtering-operators-trong-10-phut/>

- Fillter: để tiến hành lọc các phần tử được phát ra từ publisher.

- removeDuplicates: bỏ đi các phần tử liên tiếp mà giống nhau, giữ lại duy nhất một phần tử. Còn nếu các phần tử giống nhau mà không liên tiếp thì vẫn bình thường.

- compactMap: biến đổi các phần tử với kiểu giá trị này thành kiểu giá trị khác và lượt bỏ đi các giá trị không đạt theo điều kiện.

compactMap { Float($0) }

- ignoreOutPut: loại trừ hết tất cả các phần tử được phát ra. Tới lúc nhận được completion thì sẽ kết thúc.

- first(where:) tìm kiếm phần tử đầu tiên phù hợp với yêu cầu đặt ra. Sau đó sẽ tự completion.

first(where: { $0 % 2 == 0 })

- last(where:) tìm ra phần tử cuối cùng được phát đi phù hợp với điều kiện. Miễn là trước khi có completion.

last(where: { $0 % 2 == 0 })

- dropFirst: loại bỏ đi 1 số lượng các phần tử tính từ đầu.

dropFirst(3)

- drop(while:): phần tử nào thoả mãn điều kiện sẽ bị loại đi.

drop(while: { $0 % 5 != 0 })

- drop(untilOutputFrom:): chỉ phát khi thoả mãn điều kiện còn lại sẽ bị loại bỏ

drop(untilOutputFrom: isReady)

- prefix(:): ngược lại với drop

prefix(2)

- prefix(while:)

prefix(while: { $0 < 3 })

- prefix(untilOutputFrom:)

prefix(untilOutputFrom: isReady)

* 1. **Combining operators**:

<https://fxstudio.dev/combine-combining-operators-trong-10-phut/>

- prepend(Output): cung cấp trước các giá trị cho 1 publisher.

prepend(1, 2)

+ Và publisher sẽ phát các giá trị đó đi trước tiên. Sau đó mới tới các giá trị mà publisher phát.

+ kiểu dữ liệu thì trùng với kiểu input của publisher.

- prepend(Sequence): thay vì các giá trị riêng lẻ. Chúng ta sẽ ném cho nó 1 array hoặc set.

prepend([1, 2])

prepend(Set(1...2))

prepend(stride(from: 6, to: 11, by: 2))

- prepend(Publisher): sử dụng 1 publisher để chuẩn bị

- Appending: bổ sung các giá trị ra phía sau cùng cho publisher. các giá trị đó sẽ được phát sau khi publisher phát đi completion.

+ append(Output)

append(3, 4)

+ append(Sequence)

append([3, 4])

+ append(Publisher)

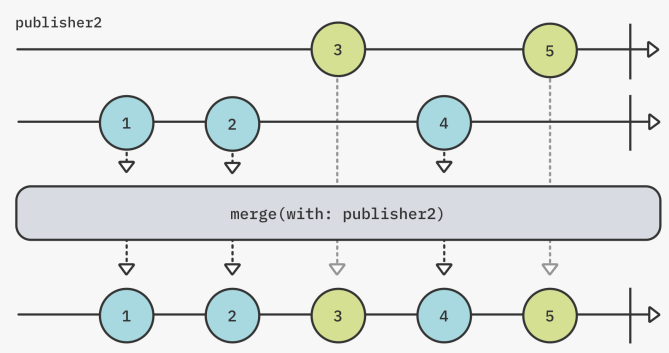
- switchToLastest: Dùng trong trường hợp kết hợp nhiều publisher. Và bạn chỉ cần publisher nào cuối cùng phát ra giá trị thì nhận nó.

- merge(with:):

merge(with: publisher2)

kết hợp nhiều publisher lại thành 1 publisher. Mỗi publisher trong đó cứ tự do phát và subcriber sẽ nhận được hết

+ khi nào cả 2 publisher kết thúc thì mới kết thúc chung.



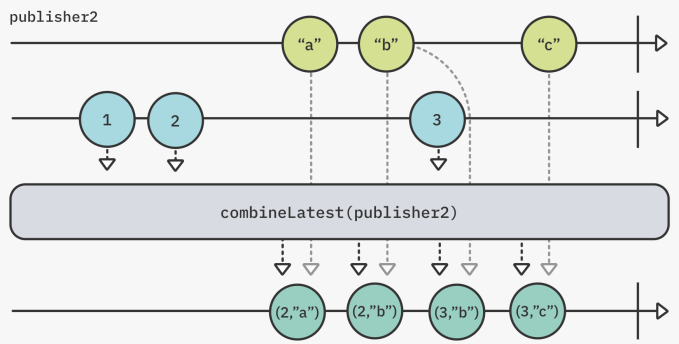
- combineLastest:

combineLatest(publisher2)

+ Kết hợp các publisher lại với nhau

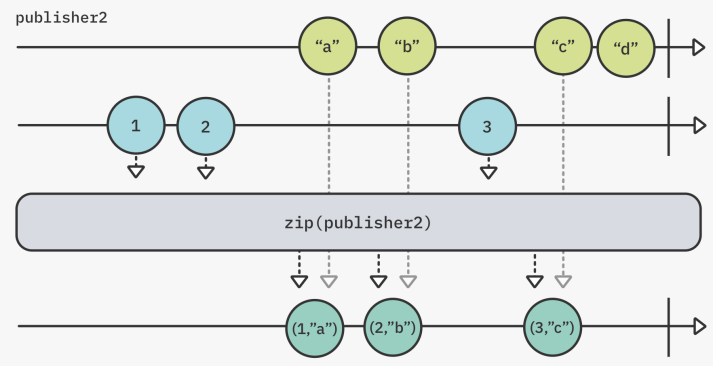
+ Nhận được một lúc cả 2 giá trị cuối cùng của mỗi publisher.

+ Giá trị nhận được là sự kết hợp từ các giá trị ở mỗi publisher.



- zip: là sự kết hợp các Output của các publisher lại (tương tự như kiểu Tuple trong Swift kết hợp nhiều kiểu giá trị lại với nhau)

+ Khi nào đủ Output thì nó sẽ phát cho subscriber các cặp giá trị.



* 1. **Time operators**:

- Delay:

+ tạo ra 1 publisher mới từ 1 publisher gốc.

+ khi publisher gốc phát đi 1 giá trị, thì sau khoảng thời gian cài đặt thì publisher delay sẽ phát cùng giá trị đó đi.

- debounce:

+ tạo ra 1 Publisher mới với gian được gán vào.

+ Khi đủ thời gian thì Publisher mới này sẽ phát ra giá trị, với gián trị là giá trị mới nhất của Publisher gốc.

- throttle:

- Timing out:

+ bạn cần set cho nó 1 thời gian. Nếu quá thời gian đó mà publisher gốc không có phát bất cứ gì ra thì publisher timeout sẽ tự động kết thúc.

+ nếu có giá trị gì mới được phát trong thời gian timeout thì sẽ tính lại từ đầu.

- Measuring time:

+ đo lường thời gian khi có sự thay đổi trên publisher

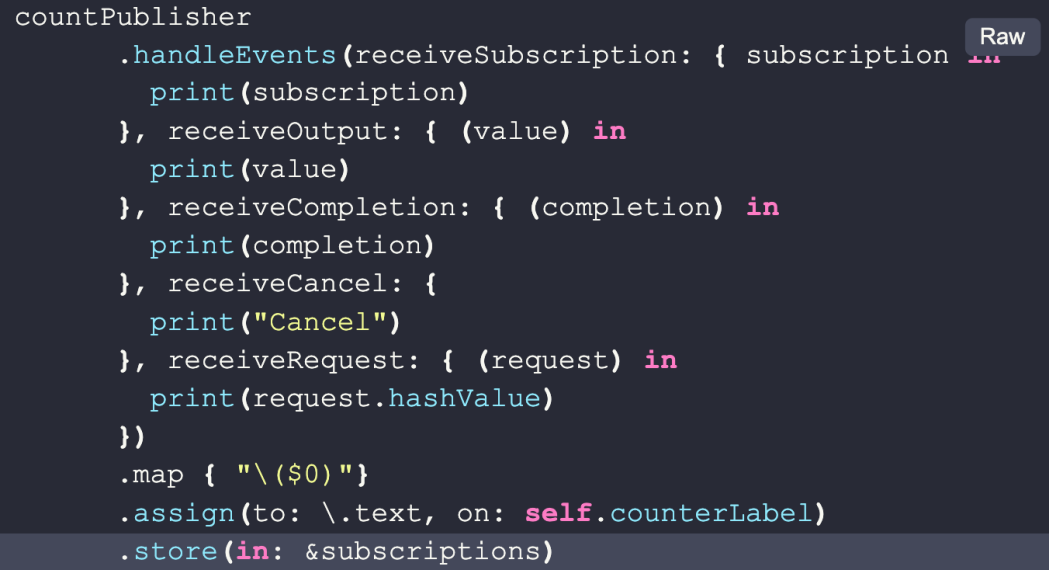
1. **Binding:**

@Published

@ObservedObject

1. **HandleEvents:**

Handle các sự kiện xảy ra của 1 Publisher.



1. **Callback:**

muốn nhận được phản hồi lại cho ViewController biết, sau khi thực hiện sự kiện nào đó.

1. **Scheduler:**

<https://magz.techover.io/2023/03/18/schedulers-in-swift-combine-framework/>

- a scheduler is a protocol that defines when and how to execute a closure.

- Scheduler Là cơ chế đồng bộ hoá của framework Combine. Nó xác định Context là gì, thực hiện công việc ở đâu (where), thực hiện công việc khi nào (when).

+ Where: Runloop, dispatchQueue, OperationQueue

+ When: Có nghĩa là thời gian ảo, tính theo thời gian của Scheduler, công việc được thực hiện bởi Scheduler phải tuân thủ theo thời gian của Scheduler, có thể không tương ứng với thời gian thực tế của hệ thống.

- Sử dụng RunLoop.main, DispatchQueue.main hoặc OperationQueue.main để thực hiện công việc liên quan UI. Không có sự khác biệt giữa chúng.

- khi bạn không chỉ định bất kì scheduler nào, Combine vẫn cung cấp cho bạn một scheduler mặc định, scheduler này sử dụng cùng thead với nơi nó được tạo ra. Ví dụ: nếu bạn bắn một sự kiện nào đó từ background thread thì bạn sẽ nhận được sự kiện đó cùng thread với nơi scheduler được tạo ra là background thread.

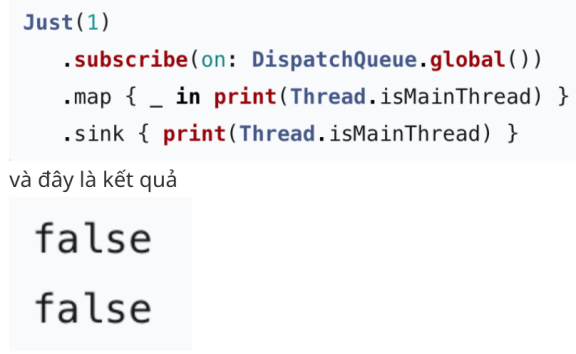
- please avoid running RunLoops in DispatchQueues, it causes some really gnarly resource usage.

- Thông thường các hoạt như call API được xử lý ở background thread để UI không bị block. Sau khi call API thì thường update lại UI và công việc này được thực hiện trên main thread. Cách thức của Combine để thực hiện việc này là schitch schedulers, sử dụng subscribe(on:) và receive(on:).

+ receive(on:) thay scheduler của tất cả những operators sau nó



+ subscribe(on:): Method này thay đổi scheduler của subscribe, cancel, and request operations. Vị trí của subscribe là không quan trọng, chỉ đơn giản là đăng ký event đó ở thread nào thôi.





- Pattern phổ biến để thực hiện asynchronous trong Combile là thực hiện subscribe ở background thread và recive ở main thread.



1. **DispatchQueue:**

<https://khorbushko.github.io/article/2020/12/05/schedulers-in-combine-DispatchQueue.html>

- Thực hiện công việc trên một dispatch queue cụ thể: serial, concurrent, main and global.

- serial, global: thường dùng cho các công việc dưới background.

- main queue: sử dụng cho việc update UI



- SchedulerOptions:

+DispatchQoS: control quality of service, priority of this thread. For example, we may specify that work should be done on .utility priority. Default - .unspecified.

+ DispatchWorkItemFlags: allow us to control the behavior of work item.

+ DispatchGroup: specify to which group of a task this task should be added. Default - nil.

- Different between RunLoop and DispatchQueue:

+ RunLoop will execute this task within next loop (so with some minimal delay) and DispatchQueue may/or may mot optimize this call and execute it immediately.

+ DispatchQueue may execute the task for a bit faster than RunLoop.main, but for a real-world example, this is almost always nothing.

+ DispatchQueue.main runs callbacks in all of the .common modes, which include the modes used when tracking touch and mouse events. If you use DispatchQueue.main, your events will be delivered while the use user in the middle of a touch or drag.

+ RunLoop.main runs callbacks only when the main run loop is running in the .default mode, which is not the mode used when tracking touch and mouse events. If you use RunLoop.main as a Scheduler, your events will not be delivered while the user is in the middle of a touch or drag.

=> Example: If you want to loading image when scroll table -> you have to using DispatchQueue.

1. **OperationQueue:**

<https://khorbushko.github.io/article/2020/12/14/schedulers-in-combine-operationQueue.html>

- it’s a queue that controls how operations can be executed.

- Remember that every task by default will be executed async concurrently on available Threads, so the order is not guaranteed.

- sử dụng OperationQueue.main cho các công việc liên quan đến UI.

- các queue khác sử dụng cho công việc background.

- không khuyến khích sử dụng operationQueue với maxConcurrentOperations lớn hơn 1.

1. **RunLoop:**

<https://khorbushko.github.io/article/2020/12/01/schedulers-in-combine-runLoop.html>

- Thực hiện công việc trên một RunLoop cụ thể.

- RunLoop scheduler associated with concrete Thread, thus Thread works with RunLoop and may create it if needed for us.

- Runloop has only one mode: default

- UIKit and AppKit run the RunLoop in the default mode when idle. But, in particular, when tracking a user interaction (like a touch or a mouse button press), they run the RunLoop in a different, non-default mode. So a Combine pipeline that uses **receive(on: RunLoop.main) will not deliver signals while the user is touching or dragging**.

- RunLoop.current.run(mode: .default, before: Date.distantFuture): to run Runloop when using runloop in a Thread.

- Make sure that RunLoop you are using is running and in expected mode.

- receive(on: RunLoop.main) will not deliver signals while the user is touching or dragging.

- there is a possible minimal delay while perform executed (usually not important)

- RunLoop is not Thread-safe - so be careful when using it.

- avoid RunLoop.current if u not sure in usage and instead use RunLoop.main or DispatchQueue

- Usage of Runloop:

+ in Timer

+ Gesture’s can’t be processed without RunLoop.

+ downloading a lot of images in some Feed - here u can also use RunLoop in default mode - check AliExpress app: when u scroll the feed, images are not loading, but when u stop, they are.

+

1. **ImmediateScheduler:**

<https://khorbushko.github.io/article/2020/11/26/schedulers-in-combine.html>

- Thực hiện các hành động đồng bộ ngay lập tức. App sẽ terminate với một fatalError nếu bạn cố gắng thực hiện delay task.

- is just a sync operation on the same thread where u create some task.

- this is the default scheduler.

1. **Networking:**

<https://khorbushko.github.io/article/2021/06/04/my-attempt-to-networking.html>

1. **Notes:**

- Sử dụng publisher của NotificationCenter.default nhằm bắt sự kiện notification textDidChangeNotification.

**- *Lấy data từ network*:**

**<https://fxstudio.dev/combine-vs-uikit-networking/>**

+ Cách truyênf thống:

1. Sử dụng đối tượng URLSession.

2.Tạo ra 1 đối tượng dataTask để thực hiện việc connect & parse khi nhận được response: decode & error handling.

+ Dùng combine:

1. Sử dụng URLSession, dùng publisher của URLSession:

*URLSession.shared.dataTaskPublisher(for: url)*

1. Tạo 1 instance của cancellable.
2. subscription tới Publisher của DataTask:

Publisher & các setup cần thiết:

+ Có được Publisher từ dataTaskPublisher

+ map để lấy phần data từ response trả về. Các dữ liệu khác không cần quan tâm

+ decode để từ JSON mà biến đổi thành đối tượng theo các class/struct đã định nghĩa

+ replaceError để handler các error. Trường hợp này nếu có lỗi thì trả về 1 array rỗng

+ eraseToAnyPublisher xoá sạch dấu vết để lại

Subscriber:

+ Sử dụng sink để bạn muốn handler nhiều hơn

+ Sử dụng assign để bạn muốn binding dữ liệu trực tiếp lên thuộc tính của đối tượng.

- Khi chúng ta phải gọi 1 lúc nhiều API trong cùng 1 ViewController:

+ tạo publisher cho từng API

+ nhóm các Publisher đó lại với nhau bằng các Combining operators: Publishers.Zip, ...

- Có 1 cái Link mà phải gọi đi gọi lại nhiều lần -> tốn tài nguyên máy: Giải pháp:

+ dùng share(): Nhưng bạn cần phải hoàn thành và setup xong tất cả các subscriber trước khi nhận được data. Điều này thì khá hên xui đó.

+ Dùng multicast:

- Lưu trữ lại

- Tạo ra 1 subject ConnectablePublisher, để phát ra giá trị: PassthroughSubject<Data, URLError>

- Cho phép nhiều subscriber subscribe vào trước khi gọi connect.

- Codable: là 1 kiểu dữ liệu để có thể map dữ liệu từ JSONDecoder thành đối tượng của struct một cách nhanh chóng.

- khi tương tác trong Project và để tránh ít va chạm xung đột với Main Thread. Thì việc tốt nhất là subscribe & receive ở một Thead khác.

- receive(on: DispatchQueue.main): đảm bảo việc nhận được dữ liệu tại ViewController sẽ ở Main Thread, nhằm update UI

1. **MVVM and Combine in project:**

<https://fxstudio.dev/combine-vs-mvvm-binding/>

<https://fxstudio.dev/combine-vs-mvvm-actions/>

- Binding from ViewModel to View

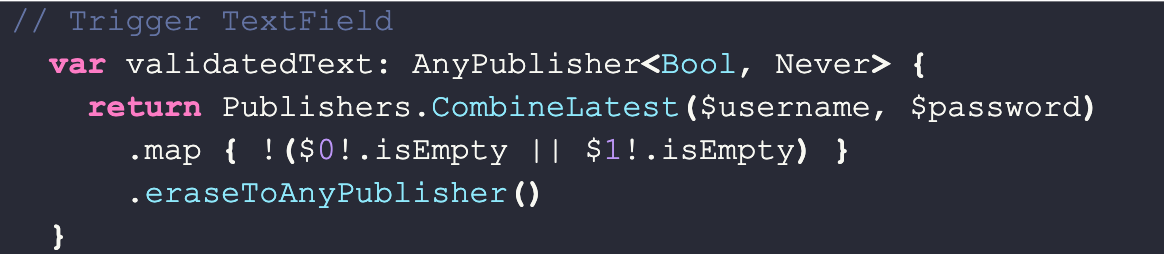
- Binding from View to ViewModel:

+ UIKit khá là bất tiện. Do bạn phải tiến hành custom hoặc viết thêm các extension cho các UI Control. Nhằm biến chúng thành nguồn phát (Publisher).

+ Nêú UIControl có các notification thì ta có thể dùng như hình bên dưới:



- Trigger: là 1 Publisher lắng nghe tới các Publisher khác. Khi đạt đủ điều kiện thì Publisher trigger sẽ phát tín hiệu đi. Ví dụ: Ta có 2 TextField là username & password. Chỉ khi nào cả 2 field đó đều có dữ liệu (không rỗng) thì Button Login mới được kích hoạt.



1. **Combine and MVVM Request:**

- thường là model call các API để lấy data.

- ViewModel liên lạc trực tiếp với Model, còn Model lại không liên lạc trực tiếp với ViewModel. Model sẽ phục vụ nhiều Controller hay View hay ViewModel nào đó. Chúng hầu như là các Data Manager, là nơi sẽ quản lý dữ

liệu tập trung. Model sẽ nhận các request tới mình để xử lý dữ liệu. => các Model có 1 đối tượng singleton của riêng mình.

- Model sẽ không phản hồi trực tiếp tới một ViewModel nào đó. Thay vì đó chúng sẽ thông báo tới 1 loạt các đối tượng đang cần dữ liệu của Model => nhiệm vụ của bạn sẽ viết các giao thức (interface) để xử lý các thông báo nhận được từ Model => Đó là mô hình Key–value observing.

1. **Reference links:**

<https://khorbushko.github.io/tags/#combine>