## What Is Combine in SwiftUI?

"The Combine framework provides a declarative Swift API for processing values over time. ... Combine declares *publishers* to expose values that can change over time, and *subscribers* to receive those values from the publishers."

Now, what are publishers and subscribers

**Publisher:** "The Publisher protocol declares a type that can deliver a sequence of values over time."

**Subscriber:** "At the end of a chain of publishers, a Subscriber acts on elements as it receives them."

So publisher exposes values on an object that can change and can be "subscribed" to or observed

One of the most common examples is the implementation of a network request handler to perform and parse the result from a server. Typically, this would require some code containing some overhead due to the validations and safeguards necessary to provide robust functionality.

```
enum NetworkError: Error {
  case invalidRequestError(String)
  case transportError(Error)
  case serverError(statusCode: Int)
  case noData
  case dataError(Error)
}
func validateName(category: String = “sport”, completion:
@escaping (Result<Data, NetworkError>) -> Void) {
  guard let url = URL(string:
"https://api.chucknorris.io/jokes/random?category=\(category)") else
{
    completion(.failure(.invalidRequestError("Invalid URL")))
    return
  }
  let networkTask = URLSession.shared.dataTask(with: url) {
data, response, error in
    if let error = error {
      completion(.failure(.transportError(error)))
      return
    }
    if let response = response as? HTTPURLResponse,
(200...299).contains(response.statusCode) == false {
      completion(.failure(.serverError(statusCode:
response.statusCode)))
      return
    }
    guard let data = data else {
      completion(.failure(.noData))
      return
    }
    do {
      completion(.success(data))
    } catch {
      completion(.failure(.dataError(error)))
    }
  }
  task.resume()
}
```

As you can see, there are a lot of validations and code that are common on network implementations.

The Combine framework makes the process of making your UI reactive much easier. But that's not all. One of the main advantages the Combine framework offers is streamlining asynchronous processes and simplifying the structure and maintainability of code.

```
enum NetworkError: Error {
  case invalidRequestError(String)
  case transportError(Error)
  case serverError(statusCode: Int)
  case noData
  case decodingError(Error)
  case encodingError(Error)
}
struct NetworkService {
    func randomQuoteFrom(category: String = “sport”)
->AnyPublisher<ChuckQuote, Error> {
        guard let url = URL(string:
"https://api.chucknorris.io/jokes/random?category=\(category)")
        else {
          return Error.invalidRequestError
         }
        return URLSession.shared
            .dataTaskPublisher(for:
EndPoint.category(category).url)
            .map(\.data)
            .decode(type: ChuckQuote.self, decoder: decoder)
            .mapError { (error) -> Error in
                switch error {
                case is URLError:
                    return
Error.addressUnreachable(EndPoint.category(category).url)
                default:
                    return Error.invalidResponse
                }
        }
        .eraseToAnyPublisher()
      }
}
```

If you're wondering what the operators in this publisher do, here's a quick rundown.

- 'Map' allows you to destructure a tuple using a key path and access just the attribute you need.
- 'Decode' decodes the data from the upstream publisher into a NameAvailableMessage instance.
- 'EraseToAnyPublisher' unwraps the result so it's not nested in multiple 'Publisher.map<>' wrappers.

```swift
let api = API()
var subscriptions = [AnyCancellable]()
api.randomQuote()
    .sink(receiveCompletion: { print($0) }, receiveValue: {
print($0) })
    .store(in: &subscriptions)
api.randomQuoteFrom("animal")
    .sink(receiveCompletion: { print($0) }, receiveValue: {
print($0) })
    .store(in: &subscriptions)
// Simulating error response
api.randomQuoteFrom("notExistingCategory")
    .sink(receiveCompletion: { print($0) }, receiveValue: {
print($0) })
    .store(in: &subscriptions)
PlaygroundPage.current.needsIndefiniteExecution = true
```

## Here's a breakdown of how the `sink` operator works

1. **Subscribe to a Publisher:** You use the sink operator to subscribe to a publisher and specify the actions to take when values are emitted, the publisher completes, or an error occurs.
2. **Closure Arguments:**

- receiveValue: This closure is called when the publisher emits a new value. It takes a single argument that represents the emitted value.
- receiveCompletion: This closure is called when the publisher either completes successfully or encounters an error. It takes a Subscribers.Completion object as an argument, which can be .finished or .failure(error). You can handle errors in this closure.

**3. Cancellation**: The sink operator returns a Cancellable object. By maintaining a reference to this object, you keep the subscription active. Releasing the reference by setting it to nil cancels the subscription.

```
import Combine

// Define a simple publisher that emits a sequence of integers
let publisher = [1, 2, 3, 4, 5].publisher

// Subscribe to the publisher using the sink operator
var cancellable: AnyCancellable?
cancellable = publisher.sink(
   receiveCompletion: { completion in
       switch completion {
       case .finished:
           print("Publisher completed successfully.")
       case .failure(let error):
           print("Error: \(error)")
       }, receiveValue: { value in
       print("Received value: \(value)")
      }
   }
)
// This keeps the subscription alive
// To release and cancel the subscription, set cancellable to
nil
cancellable = nil
```

In this example:

- We create a publisher that emits a sequence of integers.
- We use the sink operator to:
- Print each value emitted by the publisher.

- Handle the completion event, whether it completes successfully or encounters an error.

By setting cancellable to nil, we release the reference to the Cancellable object, which cancels the subscription.

## Merge, CombineLatest, and Zip: Comparing Operators of Combine for iOS

1. merge: This operator combines two publishers of the same type into one, emitting values as they arrive from either publisher. The resulting publisher emits values from both original publishers as they become available, interweaving the values in the order they're received.

2. combineLatest: This operator combines the latest values from multiple publishers whenever any of them emit a new value. The resulting publisher emits a tuple containing the latest values from each publisher every time one of them produces a new value. This is useful when you need to perform an action based on the latest values from different sources.

3. zip: This operator combines values from multiple publishers pair-wise, emitting a tuple of values when all publishers have emitted a value. The resulting publisher waits for each publisher to emit a value, then combines the values into a tuple and sends it to the subscriber. This process repeats for subsequent values. This is useful when you need to synchronize values from different publishers based on their orders.

## Merge: Bringing Publishers Together

```
let publisher1 = PassthroughSubject<Int, Never>()
let publisher2 = PassthroughSubject<Int, Never>()
let mergedPublisher = publisher1.merge(with: publisher2)
   .sink { value in
```

```
        print("Received value: \(value)")
    }
publisher1.send(1) // Output: Received value: 1
publisher2.send(2) // Output: Received value: 2
publisher1.send(3) // Output: Received value: 3
```

As you can see, mergedPublisher receives and emits values from both publisher1 and publisher2 as they arrive, combining them into a single stream. It's that easy!

## CombineLatest

This operator is all about timing. It combines the latest values from multiple publishers whenever any of them emit a new value. So, every time one publisher updates, the resulting publisher emits a tuple containing the latest values from each publisher.

```
let temperaturePublisher = PassthroughSubject<Int, Never>()
let humidityPublisher = PassthroughSubject<Int, Never>()
let weatherPublisher = Publishers.CombineLatest(temperaturePublisher,
humidityPublisher)
    .sink { temperature, humidity in
        print("Temperature: \(temperature), Humidity: \(humidity)")
    }
temperaturePublisher.send(72) // No output yet
humidityPublisher.send(45)     // Output: Temperature: 72, Humidity: 45
temperaturePublisher.send(74) // Output: Temperature: 74, Humidity: 45
```

In this case, the `weatherPublisher` combines the latest values from `temperaturePublisher` and `humidityPublisher` and emits a tuple every time one of them produces a new value. It's like magic, but better!

## Zip

This operator is all about synchronization. It combines values from multiple publishers pair-wise, emitting a tuple of values when all publishers have emitted a value.

```
let usernamePublisher = PassthroughSubject<String, Never>()
let agePublisher = PassthroughSubject<Int, Never>()
let userPublisher = Publishers.Zip(usernamePublisher, agePublisher)
    .sink { username, age in
        print("Username: \(username), Age: \(age)")
    }
usernamePublisher.send("johndoe") // No output yet
agePublisher.send(30)            // Output: Username: johndoe, Age: 30
usernamePublisher.send("janedoe") // No output yet
agePublisher.send(28)            // Output: Username: janedoe, Age: 28
```

What you have to learn

PassthroughSubject
CurrentValueSubject
Never Type