Using Combine

with Swift

Joseph Heck

Using Combine

Joseph Heck

Version 0.8, 2019-12-07

Table of Contents

About This Book

This is an intermediate to advanced book, focusing fairly narrowly on how to use the Combine framework provided by Apple. The writing and examples expect that you have a solid understanding of Swift including reference and value types, protocols, and comfort with using common elements from the Foundation framework.

If you are also just starting with Swift, Apple provides a number of resources to learn it, and there are truly amazing tutorials and introductions available in book form a number of authors, including A Swift Kickstart by Daniel Steinberg and Hacking with Swift by Paul Hudson.

This book provides a very abbreviated introduction to the concept of functional reactive programming, which is what this library is meant to provide.

Supporting this effort

This is a work in progress. Please support this effort by purchasing a copy.

The book is available online at no cost. If you find the content useful, please consider supporting the effort with a purchase of the PDF or ePub version at http://gumroad.com/l/usingcombine. The money collected for sales of this book will go to hiring editors (copyediting and technical editing), and if sufficient funds exist, hiring graphic design for diagrams.

The content for this book, including sample code and tests, are sourced from the GitHub repository: https://github.com/heckj/swiftui-notes.

To report a problem (typo, grammar, or technical fault), please Open an issue in GitHub. If you are so inclined, feel free to fork the project and send me pull requests with updates or corrections.

Acknowledgements

Thank you

Michael Critz designed and provided the cover art.

Reviews, corrections, and updates from:

Mycroft Canner, Max Desiatov, Nanu Jogi, Serhii Kyrylenko, Michel Mohrmann, Lee O'Mara, Zachary Recolan, Andrius Shiaulis, Antoine Weber, Paul Wood III, Federico Zanetello

Thank you all for taking the time and effort to submit a pull request to make this work better!

Author Bio

Joe Heck has broad software engineering development and management experience across startups and large companies. He works across all the layers of solutions, from architecture, development, validation, deployment, and operations.

Joe has developed projects ranging from mobile and desktop application development to cloud-based distributed systems. He has established teams, development processes, CI and CD pipelines, and developed validation and operational automation. Joe also builds and mentors people to learn, build, validate, deploy and run software services and infrastructure.

Joe works extensively with and in open source, contributing and collaborating with a wide variety of open source projects. He writes online across a variety of topics at https://rhonabwy.com/.

Where to get this book

The contents of this book are available as HTML.

You may also purchase the PDF or ePub at http://gumroad.com/l/usingcombine. Any purchases will support the development of this book, including technical review, copy editing, and graphics work.

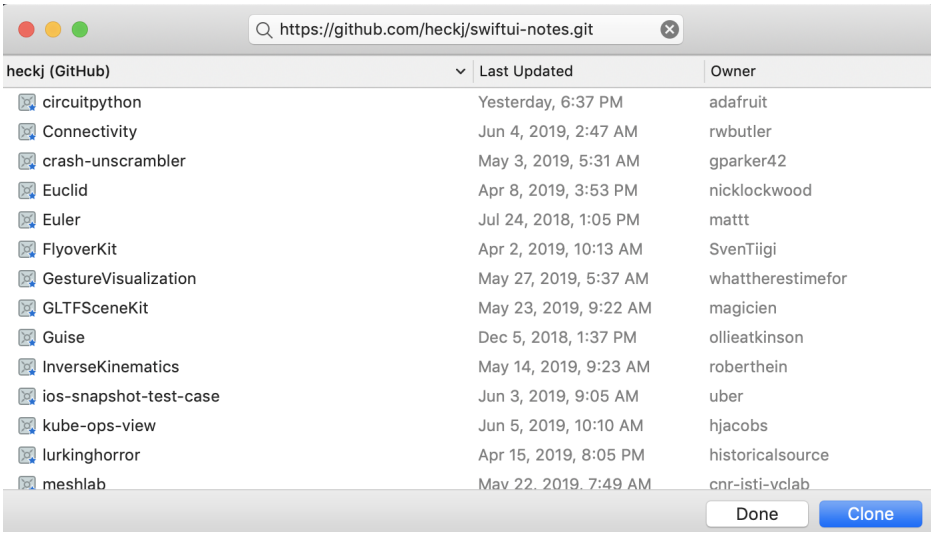
Updates of the content will be made to the online version as development continues. Larger updates and announcements will also be provided through the author's profile at Gumroad.

The contents of this book, as well as example code and unit tests referenced from the book, are link in an Xcode project (`SwiftUI-Notes.xcodeproj`), sourced from the GitHub repository heckj/swiftui-notes. The project includes unit tests illustrating the behavior of Combine and example code showing Combine integration with UIKit and SwiftUI.

Download the project

The project associated with this book requires Xcode 11 and MacOS 10.14 or later.

- From the Welcome to Xcode window, choose **Clone an existing project**
- Enter `https://github.com/heckj/swiftui-notes.git` and click `Clone`



- Choose the `master` branch to check out

Introduction to Combine

In Apple's words, Combine is:

> a declarative Swift API for processing values over time.

Combine is Apple's take on a functional reactive programming library, akin to RxSwift. RxSwift itself is a port of ReactiveX. Apple's framework uses many of the same functional reactive concepts that can be found in other languages and libraries, applying the staticly-typed nature of Swift to their solution.

If you are already familar with RxSwift there is a pretty good cheat-sheet for translating the specifics between Rx and Combine, built and inspired by the data collected at https://github.com/freak4pc/rxswift-to-combine-cheatsheet.

Another good overview is a post Combine: Where's the Beef? by Casey Liss describing how Combine maps back to RxSwift and RxCocoa's concepts, and where it is different.

Functional reactive programming

Functional reactive programming, also known as data-flow programming, builds on the concepts of functional programming. Where functional programming applies to lists of elements, functional reactive programming is applied to streams of elements. The kinds of functions in functional programming, such as `map`, `filter`, and `reduce` all have analogues that can be applied to streams. In addition, functional reactive programming includes functions to split streams, create pipelines of operations to transform the data within a stream, and merge streams.

There are many parts of the systems we program that can be viewed as asynchronous streams of information - events, objects, or pieces of data. Programming practices defined the Observer pattern for watching a single object, getting notified of changes and updates. If you view this over time, these updates make up a stream of objects. Functional reactive programming, or Combine in this case, allows you to create code that describes what happens when getting data in a stream.

You may want to create logic to watch more than one element that is changing. You may also want to include logic that does additional asynchronous operations, some of which may fail. You may also want to change the content of the streams based on timing, or change the timing of the content. Handling the flow of these event streams, the timing, errors when they happen, and coordinating how a system responds to all those events is at the heart of this kind of programming.

A solution based on functional reactive programming is particularly effective when programming user interfaces. Or more generally for creating pipelines that process data from external sources or rely on asynchronous APIs.

Combine specifics

Applying these concepts to a strongly typed language like Swift is part of what Apple has created in

Combine. Combine embeds the concept of back-pressure, which allows the subscriber to control how much information it gets at once and needs to process. In addition, it supports efficient operation with the notion of streams that are cancellable and driven primarily by the subscriber.

Combine is set up to be composed, and includes affordances to integrate existing code to incrementally support adoption.

Combine is supported by a couple of Apple's other frameworks. SwiftUI is the obvious example that has the most attention, with both subscriber and publisher elements. RealityKit also has publishers that you can use to react to events. And Foundation has a number of Combine specific additions including NotificationCenter, URLSession, and Timer as publishers.

Any asynchronous operation API *can* be leveraged with Combine. For example, you could use some of the APIs in the Vision framework, composing data flowing to it, and from it, by leveraging Combine.

> In this work, I'm going to call a set of composed operations in Combine a **pipeline**. Pipeline is not a term that Apple is (yet?) using in its documentation.

When to use Combine

Combine fits most naturally when you want to set up a something that is "immediately" reactive to a variety of inputs. User interfaces fit very naturally into this pattern.

The classic examples in functional reactive programming and user interfaces frequently show form validation, where user events such as changing text fields, taps, or mouse-clicks on UI elements make up the data being streamed. Combine takes this quite a bit further, enabling watching of properties, binding to objects, sending and receiving higher level events from UI controls, and supporting integration with almost all of Apple's existing API ecosystem.

Some things you can do with Combine include:

- You can set up pipelines to enable the button for submission only when values entered into the fields are valid.
- A pipeline can also do asynchronous actions (such as checking with a network service) and using the values returned to choose how and what to update within a view.
- Pipelines can also be used to react to a user typing dynamically into a text field and updating the user interface view based on what they're typing.

Combine is not limited to user interfaces. Any sequence of asynchronous operations can be effective as a pipeline, especially when the results of each step flow to the next step. An example of such might be a series of network service requests, followed by decoding the results.

Combine can also be used to define how to handle errors from asynchronous operations. Combine supports doing this by setting up pipelines and merging them together. One of Apple's examples with Combine include a pipeline to fall back to getting a lower-resolution image from a network service when the local network is constrained.

Many of the pipelines you create with Combine will only be a few operations. Even with just a few operations, Combine can still make it much easier to view and understand what's happening when you compose a pipeline.

Apple's Documentation

The online documentation for Combine can be found at https://developer.apple.com/documentation/combine. Apple's developer documentation is hosted at https://developer.apple.com/documentation/.

WWDC content

Apple provides video, slides, and some sample code in sessions it's developer conferences. Details on Combine are primarily from WWDC 2019.

A number of these introduce and go into some depth on Combine:

- Introducing Combine
  - PDF of presentation notes
- Combine in Practice
  - PDF of presentation notes

A number of additional WWDC19 sessions mention Combine:

- Modern Swift API Design
- Data Flow Through SwiftUI
- Introducing Combine and Advances in Foundation
- Advances in Networking, Part 1
- Building Collaborative AR Experiences
- Expanding the Sensory Experience with Core Haptics

Additional Online Combine Resources

In addition to Apple's documentation, there are a number of other online resources where you can find questions, answers, discussion, and descriptions of how Combine operates.

- The Swift Forums (hosted from the swift open source project) has a combine tag with a number of interesting threads. While the Combine framework is **not** open source, some of it's implementation and specifics are discussed in these forums.
- Stackoverflow also has a sizable (and growing) collection of Combine related Q&A.

Core Concepts

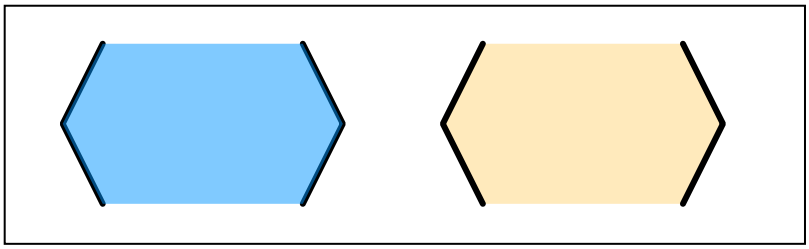
Publisher, Subscriber

Two key concepts, described in Swift with protocols, are **publisher** and **subscriber**.

When you are talking about programming (and especially with Swift and Combine), quite a lot is described by the types. When you say a function or method returns a value, that value is generally described as being "one of this type".

Combine is all about returning many possible values over time - the whole sequence of returning those values as one concept. Combine also goes farther, in that it not only talks about the types that can be returned, but the failures that might happen as well.

The first core concept to introduce in this respect is the publisher. A publisher fundamentally provides data, when available, and upon request. A publisher that hasn't had any requests will not provide any data. When you are describing a Combine publisher, you describe it with two associated types: one for Output and one for Failure.

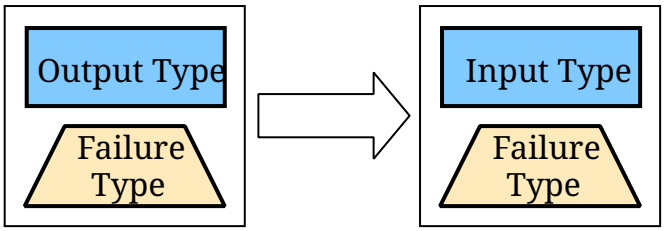


These are often written using Swift's generics syntax which uses the `<` and `>` symbol around a string. This represents that we are talking about a generic instance of this type of value. For example, if a publisher returned an instance of String, and could return a failure in the form of an instance of URLError, then the publisher might be described with the string `<String>,<URLError>`.

The corresponding concept that matches with a publisher is a subscriber, and is the second core concept to introduce. Publishers and subscribers are meant to be connected, and make up the core of Combine.

A subscriber is responsible for requesting data and accepting the data (and possible failures) provided by a publisher. In Combine, the subscriber initiates the request for data, and controls the amount of data it receives. It can be thought of as "driving the action" within Combine, as without a subscriber, the other components in Combine are idle. A subscriber is also described with two associated types, one for Input and one for Failure.

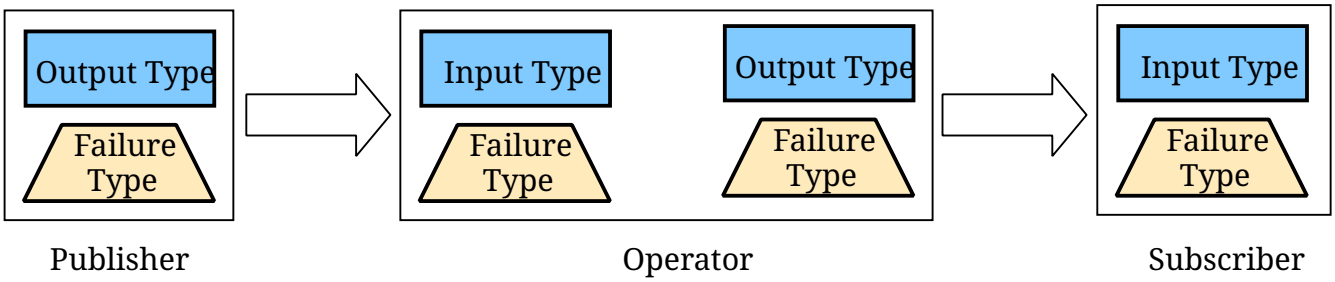
When you connect a subscriber to a publisher, both types must match: Output to Input, and Failure to Failure. One way to visualize this is as a series of operations on two types in parallel, where both types need to match in order to plug the components together.



The third core concept is an operator - an object that acts both like a subscriber and a publisher. Operators are classes that adopt both the Subscriber protocol and Publisher protocol. They support subscribing to a publisher, and sending results to any subscribers.

You can create chains of these together, for processing, reacting, and transforming the data provided by a publisher, and requested by the subscriber.

I'm calling these composed sequences **pipelines**.



Operators can be used to transform types - both the Output and Failure type. Operators may also split or duplicate streams, or merge streams together. Operators must always be aligned by the combination of Output/Failure types. The compiler will enforce the matching types, so getting it wrong will result in a compiler error (and sometimes a useful *fixit* snippet.)

A simple pipeline written in swift, using Combine, might look like:

```
let _ = Just(5) ①
    .map { value -> String in ②
        // do something with the incoming value here
        // and return a string
        return "a string"
    }
    .sink { receivedValue in ③
        // sink is the subscriber and terminates the pipeline
        print("The end result was \(receivedValue)")
    }
```

① The pipeline starts with the publisher `Just`, which responds with the value that its defined with (in this case, the Integer `5`). The output type is <Integer>, and the failure type is <Never>.

② the pipeline then has a `map` operator, which is transforming the value. In this example it is ignoring the published input and returning a string. This is also transforming the output type to <String>, and leaving the failure type still set as <Never>

③ The pipeline then ends with a `sink` subscriber.

When you are viewing a pipeline, or creating one, you can think of it as a sequence of operations linked by the types. This pattern will come in handy when you start constructing your own pipelines. When creating pipelines, you are often selecting operators to help you transform the types, to achieve your end goal. That end goal might be enabling or disabling a user interface element, or it might be retrieving some piece of data to be displayed.

Many Combine operators are specifically created to help with these transformations. Some operators require conformance to an input or failure type. Other operators may change either or both the failure and output types. For example, there are a number of operators that have a similar operator prefixed with `try`, which indicates they return an <Error> failure type.

An example of this is map and tryMap. The `map` operator allows for any combination of Output and Failure type and passes them through. `tryMap` accepts any Input, Failure types, and allows any Output type, but will always output an `<Error>` failure type.

Operators like `map` allow you to define the output type being returned by infering the type based on what you return in a closure provided to the operator. In the example above, the `map` operator is returning a String output type since that it what the closure returns.

To illustrate the example of changing types more concretely, we expand upon the logic to use the values being passed. This example still starts with a publisher providing the types `<Int>`, `<Never>` and end with a subscription taking the types `<String>`, `<Never>`.

```
let _ = Just(5) ①
    .map { value -> String in ②
        switch value {
        case _ where value < 1:
            return "none"
        case _ where value == 1:
            return "one"
        case _ where value == 2:
            return "couple"
        case _ where value == 3:
            return "few"
        case _ where value > 8:
            return "many"
        default:
            return "some"
        }
    }
    .sink { receivedValue in ③
        print("The end result was \(receivedValue)")
    }
```

① Just is a publisher that creates an `<Int>`, `<Never>` type combination, provides a single value and then completes.

② the closure provided to the `.map()` function takes in an `<Int>` and transforms it into a `<String>`.

Since the failure type of <Never> is not changed, it is passed through.

③ `sink`, the subscriber, receives the `<String>`, `<Never>` combination.

When you are creating pipelines in Xcode and don't match the types, the error message from Xcode may include a helpful *fixit*. In some cases, such as the example above, the compiler is unable to infer the return types of closure provided to `map` without specifying the return type. Xcode (11 beta 2 and beta 3) displays this as the error message: `Unable to infer complex closure return type; add explicit type to disambiguate`. In the example above, we explicitly specified the type being returned with the line `value  String in`.

You can view Combine publishers, operators, and subscribers as having two parallel types that both need to be aligned - one for the functional case and one for the error case. Designing your pipeline is frequently choosing how to convert one or both of those types and the associated data with it.

More examples, and some common tasks, are detailed in the section on patterns.

Describing pipeline components with marble diagrams

Any functional reactive pipeline can be tricky to understand. A publisher is generating and reporting data, operators are reacting to that data and potentially changing it, and subscribers accepting it. That in itself would be complicated, but some elements of Combine also may change when things happen - introducing delays, collapsing multiple values into one, and so forth. Because these can be complex to understand, the functional reactive programming community often uses a visual description of what's changing called a **marble diagram**.

As you learn about Combine, you may find yourself looking at other functional reactive programming systems, such as RxSwift or ReactiveExtensions. The documentation associated with these systems often uses marble diagrams.

These diagrams focus on describing how a specific element changes a stream of data. It shows data changing over time, as well as the timing of those changes.

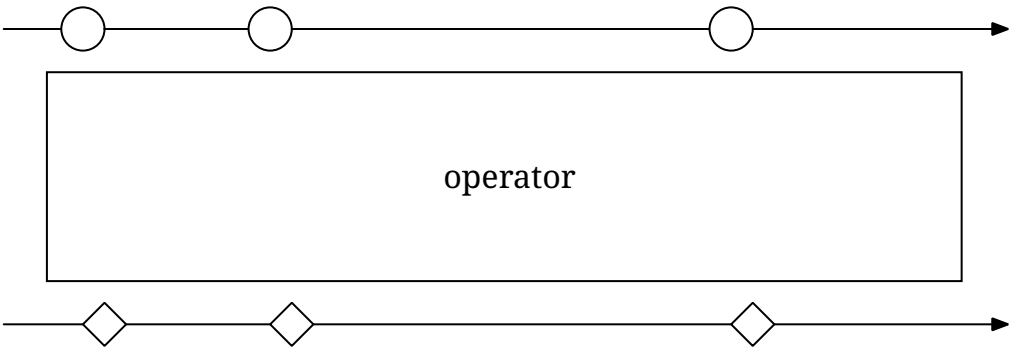


Figure 1. An example marble diagram

How to read a marble diagram:

- The diagram centers around whatever element is being described, an operator in this case. The name of the operator is often on the central block.
- The lines above and below represent data moving through time. The left is earlier and the right is later. The symbols on the line represent discrete bits of data.
- It is often assumed that data is flowing downward. With this pattern, the top line is indicating the inputs to the operator and the bottom line represents the outputs.
- In some diagrams, the symbols on the top line may differ from the symbols on the bottom line. When they are different, the diagram is typically implying that the type of the output is different from the type of the input.
- In other places, you may also see a vertical bar or an X on the timeline, or ending the timeline. That is used to indicate the end a stream. A bar at the end of a line implies the stream has terminated normally as complete. An X indicates than an error or exception was thrown.

These diagrams often ignore the setup (or teardown) of a pipeline, preferring to focus on one element to describe how that element works.

Marble diagrams for Combine

This book uses an expansion of the basic marble diagram, modified slightly to match some of the specifics of Combine. The most notable difference are two lines for input and output. Since Combine explicitly types both the input and the failure, these are represented separately.

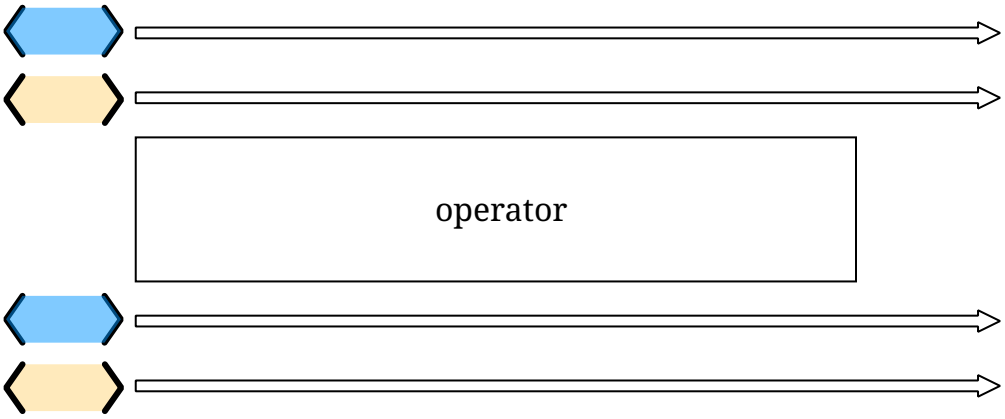


Figure 2. An expanded Combine specific marble diagram

If a publisher is being described, the two lines are below the element, following the pattern that "data flows down". An operator, which acts as both a publisher and subscriber, would have two sets - one above and one below. And a subscriber has the lines above it.

To illustrate how these diagrams relate to code, let's look at a simple example. In this case, we will focus on the map operator and how it can be described with this diagram.

```
let _ = Just(5)
    .map { value -> String in ①
        switch value {
        case _ where value < 1:
            return "none"
        case _ where value == 1:
            return "one"
        case _ where value == 2:
            return "couple"
        case _ where value == 3:
            return "few"
        case _ where value > 8:
            return "many"
        default:
            return "some"
        }
    }
    .sink { receivedValue in
        print("The end result was \(receivedValue)")
    }
```

① the closure provided to the `.map()` function takes in an `<Int>` and transforms it into a `<String>`. Since the failure type of <Never> is not changed, it is passed through.

Applying our diagram style to this code snippet. This diagram goes further than others in this book

will; we include the closure from the sample code to show how it relates to the diagram.

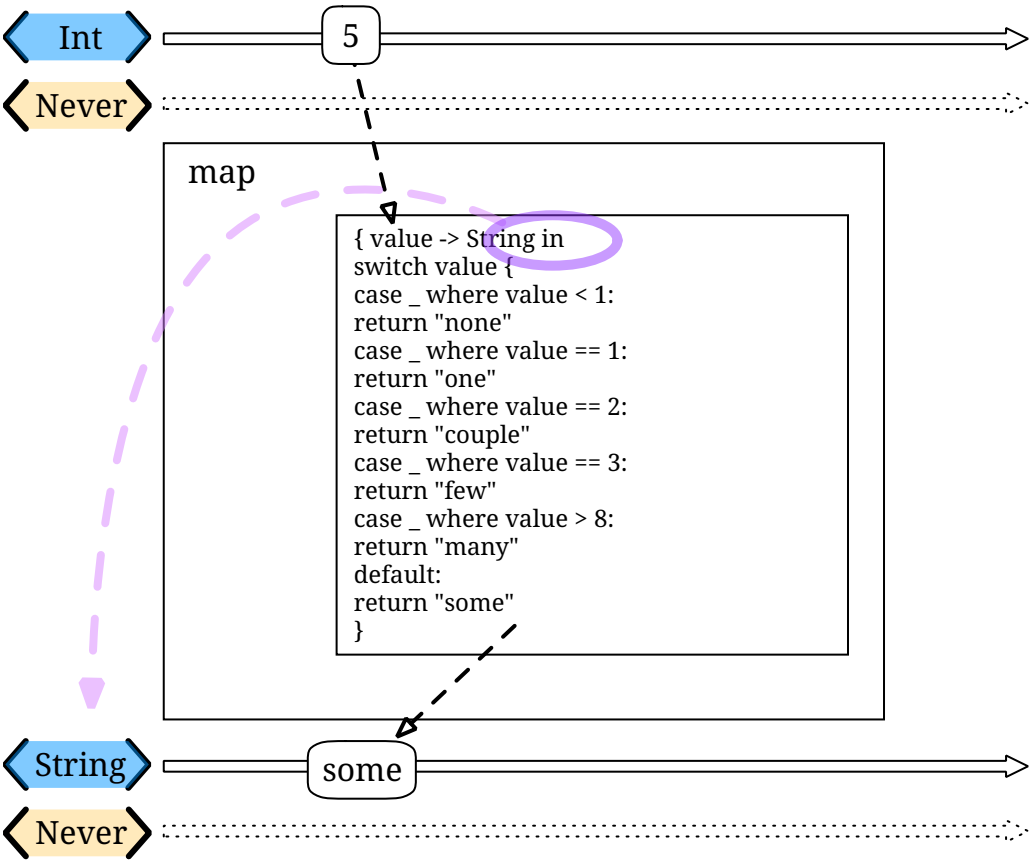


Figure 3. The example map operator from the code above:

Many combine operators are configured with code provided by you, written in a closure. Most diagrams will not include the closure or showing you how they link up. It will be implied that any code you provide through a closure to an operator in combine will be used within the box, rather than explicitly detailed as done in this diagram.

The input type for this map operator is `<Int>`, which is described with generic syntax on the top line. The failure type that is being passed to this operator is `<Never>`, described in the same syntax just below the Input type.

The map operator doesn't change or interact with the failure type, only passing it along. To represent that, the failure types - both input (above) and output (below) have been lightened.

A single input value provided (`5`) is represented on the top line. The location on the line isn't meaningful in this case, only representing that it is a single value. If multiple values were on the line, the ones on the left would be presented to the map operator before any on the right.

When it arrives, the value `5` is passed to the closure as the variable `value`. The return type of the closure (`<String>` in this case), defines the output type for the map operator. When the code within the closure completes and returns its value. In this case, the string `some` is returned for the input value `5`. The string `some` is represented on the output line directly below its input value, implying there was no explicit delay.

Most diagrams in this book won't be as complex or detailed as this example. Most of these diagrams will focus on describing what the generic operators do. This one is more complex to make it easier to understand how the diagrams can be interpretted and how they relate to any Combine code you create.

Lifecycle of Publishers and Subscribers

Combine is designed such that the subscriber ultimately controls the flow of data, and any related processing that happens in pipelines. This feature of Combine is often described using the term **back-pressure**.

This means that the subscriber drives the processing within a pipeline by providing information about how much information it wants or can accept. When a subscriber is connected to a publisher, it requests data based with a specific Demand.

In the first release of the Combine framework - in IOS 13 prior to IOS 13.2, macOS 10.15 Catalina, when the subscriber requested data with a Demand, that call itself was asynchronous. Because this process acted as the driver which triggered attached operators and ultimately the source publisher, it meant that there were scenarios where data might appear to be lost. Due to this, in IOS 13.2 and later combine releases, the process of requesting demand has been updated to a synchronous/blocking call. In practice, this means that you can be a bit more certain of having any pipelines created and full engaged prior to a publisher receiving the request to send any data.

There is an extended thread on the swift forums about this topic, if you are interested in reading the history.

The demand request is propagated up through the composed pipeline. Each operator in turn accepting the request for data and in turn requesting information from the publishers to which it is connected.

When the subscriber drives this process, it allows Combine to support cancellation. Subscribers all conform to the Cancellable protocol. This means they all have a function `cancel()` that can be invoked to terminate a pipeline and stop all related processing.

When a pipeline has been cancelled, the pipeline is not expected to be restarted. Rather than restarting a cancelled pipeline, the developer is expected to create a new pipeline.

The end to end lifecycle is enabled by subscribers and publishers communicating in a well defined sequence:

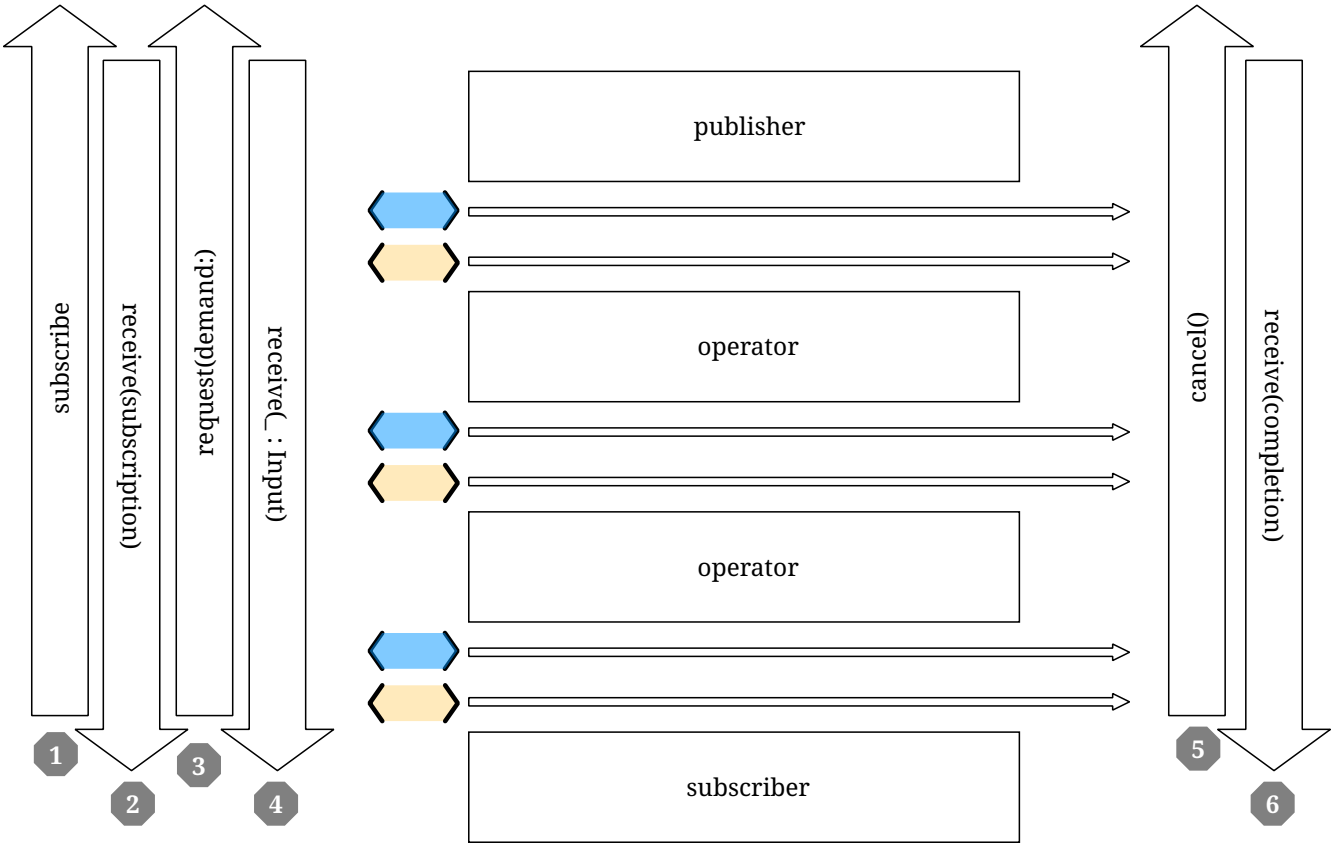


Figure 4. An The lifecycle of a combine pipeline

① When the subscriber is attached to a publisher, it starts with a call to `.subscribe(Subscriber)`.

② The publisher in turn acknowledges the subscription calling `receive(subscription)`.

③ After the subscription has been acknowledged, the subscriber requests *N* values with `request(_ : Demand)`.

④ The publisher may then (as it has values) sending *N* (or fewer) values: `receive(_ : Input)`. A publisher should never send **more** than the demand requested.

⑤ Also after the subscription has been acknowledged, the subscriber can send a cancellation with `.cancel()`

⑥ A publisher may optionally send completion: `receive(completion:)`. A completion can be either a normal termination, or may be a `.failure` completion, optionally propogating an error type. A pipeline that has been cancelled will not send any completions.

Included in the above diagram is a stacked up set of the example marble diagrams. This is to highlight where Combine marble diagrams focus in the overall lifecycle of a pipeline. Generally the diagrams infer that all of the setup has been done and data requested. The heart of a combine marble diagram is the series of events between when data was requested any completions or cancellations are triggered.

Publishers

The publisher is the provider of data. The publisher protocol has a strict contract returning values when asked from subscribers, and possibly terminating with an explicit completion enumeration.

Just and Future are extremely common sources to start your own publisher from a value or

function.

Many publishers will immediately provide data when requested by a subscriber. In some cases, a publisher may have a separate mechanism to enable it to return data. This is codified by the protocol ConnectablePublisher. A publisher conforming to `ConnectablePublisher` will have an additional mechanism to start the flow of data after a subscriber has provided a request. This could be `.autoconnect()`, which will start the flow of data as soon as a subscriber requests it. The other option is a separate `.connect()` call on the publisher itself.

Combine provides a number of additional convenience publishers:

Just	Future	@Published
Empty	Sequence	Fail
Deferred	ObservableObjectPublisher	

A number of Apple APIs outside of Combine provide publishers as well.

- SwiftUI provides `@ObservedObject` which can be used to create a publisher.
- Foundation
  - URLSession.dataTaskPublisher
  - .publisher on KVO instance
  - NotificationCenter
  - Timer

Operators

Operators are a convenient name for a number of pre-built functions that are included under Publisher in Apple's reference documentation. These functions are all meant to be composed into pipelines. Many will accept one of more closures from the developer to define the business logic of the operator, while maintaining the adherance to the publisher/subscriber lifecycle.

Some operators support bringing together outputs from different pipelines, or splitting to send to multiple subscribers. Operators may also have constraints on the types they will operate on. Operators can also help with error handling and retry logic, buffering and prefetch, controlling timing, and supporting debugging.

Mapping elements		
scan	tryScan	setFailureType
map	tryMap	flatMap

Filtering elements		
compactMap	tryCompactMap	replaceEmpty
filter	tryFilter	replaceError

removeDuplicates	tryRemoveDuplicates	

Reducing elements		
collect	collectByCount	collectByTime
reduce	tryReduce	ignoreOutput

Mathematic opertions on elements		
comparison	tryComparison	count

Applying matching criteria to elements		
allSatisfy	tryAllSatisfy	contains
containsWhere	tryContainsWhere	

Applying sequence operations to elements		
firstWhere	tryFirstWhere	first
lastWhere	tryLastWhere	last
dropWhile	tryDropWhile	dropUntilOutput
concatenate	drop	prefixUntilOutput
prefixWhile	tryPrefixWhile	output

Combining elements from multiple publishers		
combineLatest	merge	zip

Handling errors		
catch	tryCatch	assertNoFailure
retry	mapError	

Adapting publisher types		
switchToLatest	eraseToAnyPublisher	

Controlling timing		
debounce	delay	measureInterval
throttle	timeout	

Encoding and decoding		
encode	decode	

Working with multiple subscribers		
multicast		

Debugging		
breakpoint	handleEvents	print

Subjects

Subjects are a special case of publisher that also adhere to the `subject` protocol. This protocol requires subjects to have a `.send()` method to allow the developer to send specific values to a subscriber (or pipeline).

Subjects can be used to "inject" values into a stream, by calling the subject's `.send()` method. This is useful for integrating existing imperative code with Combine.

A subject can also broadcast values to multiple subscribers. If multiple subscribers are connected to a subject, it will fan out values to the multiple subscribers when `send()` is invoked. A subject is also frequently used to connect or cascade multiple pipelines together, especially to fan out to multiple pipelines.

A subject does not blindly pass through the demand from its subscribers, instead it provides a sort of aggregation point for demand. A subject will not signal for demand to its connected publishers until it has received at least one subscriber itself. When it receives any demand, it then signals for `unlimited` demand to connected publishers. With the subject supporting multiple subscribers, any subscribers that have not requested data with a demand are not provided the data until they do.

There are two types of built-in subjects with Combine: currentValueSubject and passthroughSubject. They act similiarly, the primary difference being currentValueSubject remembers and provides an initial state value for any subscribers, where passthroughSubject does not. Both will provide updated values to any subscribers when `.send()` is invoked.

Both CurrentValueSubject and PassthroughSubject are also useful for creating publishers from objects conforming to the `ObservableObject`. This protocol is supported by a number of declarative components within SwiftUI.

Subscribers

While `subscriber` is the protocol used to receive data throughout a pipeline, *the Subscriber* typically refers to the end of a pipeline.

There are two subscribers built-in to Combine: assign and sink.

Subscribers can support cancellation, which terminates a subscription and shuts down all the stream processing prior to any Completion sent by the publisher. Both `Assign` and `Sink` conform to the cancellable protocol.

`assign` applies values passed down from the publisher to an object defined by a keypath. The keypath is set when the pipeline is created. An example of this in Swift might look like:

```
.assign(to: \.isEnabled, on: signupButton)
```

`sink` accepts a closure that receives any resulting values from the publisher. This allows the developer to terminate a pipeline with their own code. This subscriber is also extremely helpful when writing unit tests to validate either publishers or pipelines. An example of this in Swift might look like:

```
.sink { receivedValue in
    print("The end result was \(String(describing: receivedValue))")
}
```

Most other subscribers are part of other Apple frameworks. For example, nearly every control in SwiftUI can act as a subscriber. The View protocol in SwiftUI defines an `.onReceive(publisher)` function to use views as a subscriber. The `onReceive` function takes a closure akin to `sink` that can manipulate `@State` or `@Bindings` within SwiftUI.

An example of that in SwiftUI might look like:

```
struct MyView : View {

    @State private var currentStatusValue = "ok"
    var body: some View {
        Text("Current status: \(currentStatusValue)")
            .onReceive(MyPublisher.currentStatusPublisher) { newStatus in
                self.currentStatusValue = newStatus
            }
    }
}
```

For any type of UI object (UIKit, AppKit, or SwiftUI), assign can be used with pipelines to manipulate properties.

When you are storing a reference to your own subscriber in order to clean up later, you generally want a reference to cancel the subscription. anyCancellable provides a type-erased reference that converts any subscriber to the type AnyCancellable, allowing the use of `.cancel()` on that reference, but not access to the subscription itself (which could, for instance, request more data).

Developing with Combine

Developing with Combine can take any of a number of different forms.

A common starting point is composing pipelines, leveraging existing publishers, operators, and subcribers. A number of examples within this book highlight various patterns, many of which are aimed at providing declarative responses to user inputs within interfaces.

You may also want to create APIs that integrate more easily into Combine. For example, creating a publisher that encapsulates a remote API, returning a single result or a series of results.

Or you might be creating a subscriber to consume and process data over time.

Reasoning about pipelines

When developing with Combine, there are two broader patterns of how publishers are used that frequently recur: one-shot and continuous.

The first is what I'm calling a "one-shot" publisher or pipeline. These publishers are expected to create a single response (or perhaps no response) and then terminate normally.

The second is what I'm calling a "continuous" publisher. These publishers, or more often pipelines, are expected to be always active and providing the means to respond to ongoing events, either user initiated or another source. In this case, the lifetime of the pipeline is significant longer, and it is often not desirable to have such pipelines fail or terminate.

When you are thinking about your development and how to use Combine, it is often beneficial to think about pipelines as being one of these types, and mixing them together to achieve your goals.

For example, the pattern Using flatMap with catch to handle errors in this book explicitly uses one-shot pipelines to support error handling on a continual pipeline.

When you are creating an instance of a publisher or a pipeline, it is worthwhile to be thinking about how you want it to work - to either be a one-shot, or continual. This choice will often inform if you do with error handling or if you want to deal with operators that manipulate the timing of the events (such as debounce or throttle).

In addition to how much data the pipeline or publisher will provide, and the expectations that come from that, you will often want to think concretely about what type the pipeline provides. Quite a number of pipelines are more about transforming data through various types, and handling possible error conditions in that processing. A good example of this is the use of returning a pipeline returning a list in the pattern Declarative UI updates from user input to provide a means to represent an "empty" result, even though the list is never expected to have more than 1 item within it.

Ultimately, using Combine is grounded at both ends by the originating publisher, and how it is providing data (when it is available), and the subscriber ultimately consuming the data.

Swift types and exposing pipelines or subscribers

When you compose pipelines within Swift, the chaining of functions results in the type being exposed as nesting generic types. If you are creating a pipeline, and then wanting to provide that as an API to another part of your code, the type definition for the exposed property or function can be exceptionally complex.

To illustrate the exposed type complexity, if you created a publisher from a PassthroughSubject such as:

```
let x = PassthroughSubject<String, Never>()
    .flatMap { name in
        return Future<String, Error> { promise in
            promise(.success(""))
            }.catch { _ in
                Just("No user found")
            }.map { result in
                return "\(result) foo"
        }
}
```

The resulting type is:

```
Publishers.FlatMap<Publishers.Map<Publishers.Catch<Future<String, Error>, Just<
String>>, String>, PassthroughSubject<String, Never>>
```

When you want to expose the code, all of that composition detail can be very distracting and make your code harder to use.

To clean up that interface, and provide a nice API boundary, there are type erased classes which can wrap either publishers or subscribers. These explicitly hide the type complexity that builds up from chained functions in Swift.

The two classes used to expose simplified types for subscribers and publishers are:

- AnySubscriber
- AnyPublisher

Most publishers also include a convenience method `eraseToAnyPublisher` that returns an instance of AnyPublisher. `eraseToAnyPublisher` is used very much like an operator, often as the last element in a chained pipeline, to simplify the type returned.

If you updated the above code to add .eraseToAnyPublisher() at the end of the pipeline:

```
let x = PassthroughSubject<String, Never>()
    .flatMap { name in
        return Future<String, Error> { promise in
            promise(.success(""))
            }.catch { _ in
                Just("No user found")
            }.map { result in
                return "\(result) foo"
        }
}.eraseToAnyPublisher()
```

The resulting type would simplify to:

```
AnyPublisher<String, Never>
```

This same technique can be immensely useful when constructing smaller pipelines within closures. For example, when you want to return a publisher in the closure for a flatMap operator, you can have a much simpler time reasoning about types by explicitly asserting the closure should expect `AnyPublisher` with the relevant associated types. An example of this can be seen in Sequencing operations with Combine.

Pipelines and threads

Combine is not just a single threaded construct. Operators, as well as publishers, can run on different dispatch queues or runloops. Composed pipelines can run across a single queue, or transfer across a number of queues or threads.

Combine allows for publishers to specify the scheduler used when either receiving from an upstream publisher (in the case of operators), or when sending to a downstream subscriber. This is critical when working with a subscriber that updates UI elements, as that should always be called on the main thread.

You may see this in code as an operator, for example:

```
.receive(on: RunLoop.main)
```

A number of operators can impact what thread or queue is being used to do the relevant processing. receive and subscribe are the two most common, explicitly moving execution of operators after and prior to their invocation respectively.

A number of additional operators have parameters that include a scheduler. Examples include delay, debounce, and throttle. These also have an impact on the queue executing the work - both for themselves and then any operators following in a pipeline. These operators all take a `scheduler` parameter, which switches to the relevant thread or queue to do the work. Any operators following them will also be invoked on their scheduler, giving them an impact somewhat like receive.

If you want to be explicit about which thread context an operator or subsequent operation will run within, define it with the receive operator.

Developing to use Combine

There are two common paths to developing code leveraging Combine.

- First is simply leveraging synchronous (blocking) calls within a closure to one of the common operators. The two most prevelant operators leveraged for this are map and tryMap, for when your code needs to throw an Error.
- Second is integrating your own code that is asynchronous, or APIs that provide a completion callback. If the code you are integrating is asynchronous, then you can't (quite) as easily use it within a closure. You need to wrap the asynchronous code with a structure that the Combine operators can work with and invoke. In practice, this often implies creating a call that returns a publisher instance, and then using that within the pipeline.

The Future publisher was specifically created to support this kind of integration, and the pattern Wrapping an asynchronous call with a Future to create a one-shot publisher shows an example.

If you want to use data provided by a publisher as a parameter or input to creating this publisher, there are two common means of enabling this:

1. Using the flatMap operator, using the data passed in to create or return a Publisher instance. This is a variation of the pattern illustrated in Using flatMap with catch to handle errors.
2. Alternately, map or tryMap can be used to create an instance of a publisher, followed immediately by chaining switchToLatest to resolve that publisher into a value (or values) to be passed within the pipeline.

The patterns Cascading UI updates including a network request and Declarative UI updates from user input illustrate these patterns.

You may find it worthwhile to create objects which return a publisher. Often this enables your code to encapsulate the details of communicating with a remote or network based API. These can be developed using URLSession.dataTaskPublisher or your own code. A simple example of this is detailed in the pattern Cascading UI updates including a network request.

Patterns and Recipes

Included are a series of patterns and examples of Publishers, Subscribers, and pipelines. These examples are meant to illustrate how to use the Combine framework to accomplish various tasks.

Since this is a work in progress: if you have a suggestion for a pattern or recipe, I'm happy to consider it.

Please Open an issue in GitHub to request something.

Creating a subscriber with sink

Goal

- To receive the output, and the errors or completion messages, generated from a publisher or through a pipeline, you can create a subscriber with sink.

References

- sink

See also

- Creating a subscriber with assign
- Testing a publisher with XCTestExpectation
- Testing a subscriber with scheduled sends from PassthroughSubject

Code and explanation

Sink creates an all-purpose subscriber to capture or react the data from a Combine pipeline, while also supporting cancellation and the publisher subscriber lifecycle.

simple sink

```
let cancellablePipeline = publishingSource.sink { someValue in ①
    // do what you want with the resulting value passed down
    // be aware that depending on the data type being returned, you may get this
closure invoked
    // multiple times.
    print(".sink() received \(someValue)")
})
```

① The simple version of a sink is very compact, with a single trailing closure that only receives data when presented through the pipeline.

sink with completions and data

```
let cancellablePipeline = publishingSource.sink(receiveCompletion: { completion in ①
    switch completion {
    case .finished:
        // no associated data, but you can react to knowing the request has been
completed
        break
    case .failure(let anError):
        // do what you want with the error details, presenting, logging, or hiding as
appropriate
        print("received the error: ", anError)
        break
    }
}, receiveValue: { someValue in
    // do what you want with the resulting value passed down
    // be aware that depending on the data type being returned, you may get this
closure invoked
    // multiple times.
    print(".sink() received \(someValue)")
})

cancellablePipeline.cancel() ②
```

① Sinks are created by chaining the code from a publisher or pipeline, and terminate the pipeline. When the sink is created or invoked on a publisher, it implicitly starts the lifecycle with the `subscribe` and will request unlimited data.

② Creating a sink is cancellable subscriber, so at any time you can take the reference that terminated with sink and invoke `.cancel()` on it to invalidate and shut down the pipeline.

Creating a subscriber with assign

Goal

- To use the results of a pipeline to set a value, often a property on a user interface view or control, but any KVO compliant object can be the target

References

- assign
- receive

See also

- Creating a subscriber with sink

Code and explanation

Assign is a subscriber that's specifically designed to apply data from a publisher or pipeline into a property, updating that property whenever it receives data. Like sink, it activates when created and requests an unlimited data updates. Assign requires the failure type to be specified as `<Never>`, so if your pipeline could fail (such as using an operator like tryMap) you will need to convert or handle the failure cases before using `.assign`.

simple assign

```
let cancellablePipeline = publishingSource ①
    .receive(on: RunLoop.main) ②
    .assign(to: \.isEnabled, on: yourButton) ③

cancellablePipeline.cancel() ④
```

① `.assign` is typically chained onto a publisher when you create it, and the return value is cancellable.

② If `.assign` is being used to update a user interface element, you need to make sure that it is being updated on the main thread. This call makes sure the subscriber is received on the main thread.

③ Assign references the property being updated using a key path, and a reference to the object being updated.

④ At any time you can cancel to terminate and invalidate pipelines with cancel(). Frequently, you cancel the pipelines when you deactivate the objects (such as a viewController) that are getting updated from the pipeline.

Making a network request with dataTaskPublisher

Goal

- One of the common use cases is requesting JSON data from a URL and decoding it.

References

- URLSession.dataTaskPublisher
- map
- decode
- sink
- subscribe

See also

- Stricter request processing with dataTaskPublisher
- Using catch to handle errors in a one-shot pipeline
- Retrying in the event of a temporary failure

Code and explanation

> This can be readily accomplished with Combine using URLSession.dataTaskPublisher followed by a series of operators that process the data. Minimally, this is map and decode before going into your subscriber.

dataTaskPublisher on URLSession.

The simplest case of using this might be:

```
let myURL = URL(string: "https://postman-echo.com/time/valid?timestamp=2016-10-10")
// checks the validity of a timestamp - this one returns {"valid":true}
// matching the data structure returned from https://postman-echo.com/time/valid
fileprivate struct PostmanEchoTimeStampCheckResponse: Decodable, Hashable { ①
    let valid: Bool
}

let remoteDataPublisher = URLSession.shared.dataTaskPublisher(for: myURL!) ②
    // the dataTaskPublisher output combination is (data: Data, response: URLResponse)
    .map { $0.data } ③
    .decode(type: PostmanEchoTimeStampCheckResponse.self, decoder: JSONDecoder()) ④

let cancellableSink = remoteDataPublisher
    .sink(receiveCompletion: { completion in
            print(".sink() received the completion", String(describing: completion))
            switch completion {
                case .finished: ⑤
                    break
                case .failure(let anError): ⑥
                    print("received error: ", anError)
            }
    }, receiveValue: { someValue in ⑦
        print(".sink() received \(someValue)")
    })
```

① Commonly you'll have a struct defined that supports at least Decodable (if not the full Codable protocol). This struct can be defined to only pull the pieces you're interested in from the JSON provided over the network.

② dataTaskPublisher is instantiated from URLSession. You can configure your own options on URLSession, or use the general shared session as you require.

③ The data that is returns down the pipeline is a tuple: `(data: Data, response: URLResponse)`. The map operator is used to get the data and drop the URL response, returning just Data down the pipeline.

④ decode is used to load the data and attempt to transform it into the struct defined. Decode can throw an error itself if the decode fails. If it succeeds, the object passed down the pipeline will be the struct from the JSON data.

⑤ If the decoding happened without errors, the finished completion will be triggered, and the value will also be passed to the receiveValue closure.

⑥ If the a failure happened (either with the original network request or the decoding), the error will be passed into with the `.failure` completion.

⑦ Only if the data succeeded with request and decoding will this closure get invoked, and the data format received will be an instance of the struct `PostmanEchoTimeStampCheckResponse`.

Stricter request processing with dataTaskPublisher

Goal

- When URLSession makes a connection, it only reports an error if the remote server doesn't respond. You may want to consider a number of responses, based on status code, to be errors. To accomplish this, you can use tryMap to inspect the http response and throw an error in the pipeline.

References

- URLSession.dataTaskPublisher
- tryMap
- decode
- sink
- subscribe

See also

- Making a network request with dataTaskPublisher
- Using catch to handle errors in a one-shot pipeline
- Retrying in the event of a temporary failure

Code and explanation

To have more control over what is considered a failure in the URL response, use a `tryMap` operator on the tuple response from dataTaskPublisher. Since dataTaskPublisher returns both the response data and the URLResponse into the pipeline, you can immediately inspect the response and throw an error of your own if desired.

An example of that might look like:

```
let myURL = URL(string: "https://postman-echo.com/time/valid?timestamp=2016-10-10")
// checks the validity of a timestamp - this one returns {"valid":true}
// matching the data structure returned from https://postman-echo.com/time/valid
fileprivate struct PostmanEchoTimeStampCheckResponse: Decodable, Hashable {
    let valid: Bool
}
enum TestFailureCondition: Error {
    case invalidServerResponse
}


let remoteDataPublisher = URLSession.shared.dataTaskPublisher(for: myURL!)
    .tryMap { data, response -> Data in ①
                guard let httpResponse = response as? HTTPURLResponse, ②
                    httpResponse.statusCode == 200 else { ③
                        throw TestFailureCondition.invalidServerResponse ④
                }
                return data ⑤
    }
    .decode(type: PostmanEchoTimeStampCheckResponse.self, decoder: JSONDecoder())

let cancellableSink = remoteDataPublisher
    .sink(receiveCompletion: { completion in
            print(".sink() received the completion", String(describing: completion))
            switch completion {
                case .finished:
                    break
                case .failure(let anError):
                    print("received error: ", anError)
            }
    }, receiveValue: { someValue in
        print(".sink() received \(someValue)")
    })
```

Where the previous pattern used a map operator, this uses tryMap, which allows us to identify and throw errors in the pipeline based on what was returned.

① tryMap still gets the tuple of `(data: Data, response: URLResponse)`, and is defined here as returning just the type of Data down the pipeline.

② Within the closure for tryMap, we can cast the response to HTTPURLResponse and dig deeper into it, including looking at the specific status code.

③ In this case, we want to consider **anything** other than a 200 response code as a failure. HTTPURLResponse.status_code is an Int type, so you could also have logic such as `httpResponse.statusCode > 300`.

④ If the predicates aren't met, then we can throw an instance of an error of our choosing, `invalidServerResponse` in this case.

⑤ If no error has occured, then we simply pass down Data for further processing.

When an error is triggered on the pipeline, a `.failure` completion is sent with the error encapsulated within it, regardless of where it happened in the pipeline.

Wrapping an asynchronous call with a Future to create a one-shot publisher

Goal

- Using Future to turn an asynchronous call into publisher to use the result in a Combine pipeline.

References

- Future

See also

- Creating a repeating publisher by wrapping a delegate based API

Code and explanation

```
import Contacts
let futureAsyncPublisher = Future<Bool, Error> { promise in ①
    CNContactStore().requestAccess(for: .contacts) { grantedAccess, err in ②
        // err is an optional
        if let err = err { ③
            promise(.failure(err))
        }
        return promise(.success(grantedAccess)) ④
    }
}.eraseToAnyPublisher()
```

① Future itself has you define the return types and takes a closure. It hands in a Result object matching the type description, which you interact.

② You can invoke the async API however is relevant, including passing in its required closure.

③ Within the completion handler, you determine what would cause a failure or a success. A call to `promise(.failure(<FailureType>))` returns the failure.

④ Or a call to `promise(.success(<OutputType>))` returns a value.

If you want to return a resolved promise as a Future publisher, you can do so by immediately returning the result you desire within Future's closure.

The following example returns a single value as a success, with a boolean `true` value. You could just as easily return `false`, and the publisher would still act as a successful promise.

```
let resolvedSuccessAsPublisher = Future<Bool, Error> { promise in
    promise(.success(true))
}.eraseToAnyPublisher()
```

An example of returning a Future publisher that immediately resolves as an error:

```
enum ExampleFailure: Error {
    case oneCase
}

let resolvedFailureAsPublisher = Future<Bool, Error> { promise in
    promise(.failure(ExampleFailure.oneCase))
}.eraseToAnyPublisher()
```

Sequencing operations with Combine

Goal

- To explicitly order asynchronous operations with a Combine pipeline

References

- Future
- flatMap
- zip
- sink

See also

- Creating a repeating publisher by wrapping a delegate based API
- The ViewController with this code is in the github project at UIKit-Combine/AsyncCoordinatorViewController.swift

Code and explanation

Any asynchronous (or synchronous) set of tasks that need to happen in a specific order can also be coordinated using a Combine pipeline. By using Future operator, the very act of completing an asynchronous call can be captured, and sequencing operators provides the structure of that coordination.

For example, by wrapping any asynchronous API calls with the Future publisher and then chaining them together with the flatMap operator, you invoke the wrapped asynchronous API calls in the order of the pipeline. Multiple parallel asynchronous efforts can be created by creating multiple pipelines, with Future or another publisher. You can wait for all parallel pipelines to complete before continuing by merging them together with the zip operator.

If you want force an Future publisher to not be invoked until another has completed, then creating the future publisher in the flatMap closure causes it to wait to be created until a value has been passed to the flatMap operator.

These techniques can be composed, creating any structure of parallel or serial tasks.

This technique of coordinating asynchronous calls can be especially effective if later tasks need data from earlier tasks. In those cases, the data results needed can be passed directly the pipeline.

An example of this sequencing follows below. In this example, buttons (arranged visually to show the ordering of actions) are highlighted when they complete. The whole sequence is triggered by a seperate button action, which also resets the state of all the buttons and cancels any existing running sequence if it's not yet finished. In this example, the asynchronous API call is a call that simply takes a random amount of time to complete to provide an example of how the timing works.

The workflow that is created is represented in steps:

- step 1 runs first.
- step 2 has three parallel efforts, running after step 1 completes.

- step 3 waits to start until all three elements of step 2 complete.
- step 4 runs after step 3 has completed.

Additionally, there is an activity indicator that is triggered to start animating when the sequence begins, stopping when step 4 has run to completion.

UIKit-Combine/AsyncCoordinatorViewController.swift

```
import UIKit
import Combine

class AsyncCoordinatorViewController: UIViewController {

    @IBOutlet weak var startButton: UIButton!

    @IBOutlet weak var step1_button: UIButton!
    @IBOutlet weak var step2_1_button: UIButton!
    @IBOutlet weak var step2_2_button: UIButton!
    @IBOutlet weak var step2_3_button: UIButton!
    @IBOutlet weak var step3_button: UIButton!
    @IBOutlet weak var step4_button: UIButton!
    @IBOutlet weak var activityIndicator: UIActivityIndicatorView!

    var cancellable: AnyCancellable?
    var coordinatedPipeline: AnyPublisher<Bool, Error>?

    @IBAction func doit(_ sender: Any) {
        runItAll()
    }

    func runItAll() {
        if self.cancellable != nil { ①
            print("Cancelling existing run")
            cancellable?.cancel()
            self.activityIndicator.stopAnimating()
        }
        print("resetting all the steps")
        self.resetAllSteps() ②
        // driving it by attaching it to .sink
        self.activityIndicator.startAnimating() ③
        print("attaching a new sink to start things going")
        self.cancellable = coordinatedPipeline? ④
            .print()
            .sink(receiveCompletion: { completion in
                print(".sink() received the completion: ", String(describing:
completion))
                self.activityIndicator.stopAnimating()
            }, receiveValue: { value in
                print(".sink() received value: ", value)
            })
    }
```

```
    // MARK: - helper pieces that would normally be in other files

    // this emulates an async API call with a completion callback
    // it does nothing other than wait and ultimately return with a boolean value
    func randomAsyncAPI(completion completionBlock: @escaping ((Bool, Error?) ->
Void)) {
        DispatchQueue.global(qos: .background).async {
            sleep(.random(in: 1...4))
            completionBlock(true, nil)
        }
    }

    /// Creates and returns pipeline that uses a Future to wrap randomAsyncAPI, then
updates a UIButton to represent
    /// the completion of the async work before returning a boolean True
    /// - Parameter button: button to be updated
    func createFuturePublisher(button: UIButton) -> AnyPublisher<Bool, Error> { ⑤
        return Future<Bool, Error> { promise in
            self.randomAsyncAPI() { (result, err) in
                if let err = err {
                    promise(.failure(err))
                }
                promise(.success(result))
            }
        }
        .receive(on: RunLoop.main)
            // so that we can update UI elements to show the "completion"
            // of this step
        .map { inValue -> Bool in ⑥
            // intentially side effecting here to show progress of pipeline
            self.markStepDone(button: button)
            return true
        }
        .eraseToAnyPublisher()
    }

    /// highlights a button and changes the background color to green
    /// - Parameter button: reference to button being updated
    func markStepDone(button: UIButton) {
        button.backgroundColor = .systemGreen
        button.isHighlighted = true
    }

    func resetAllSteps() {
        for button in [self.step1_button, self.step2_1_button, self.step2_2_button,
self.step2_3_button, self.step3_button, self.step4_button] {
            button?.backgroundColor = .lightGray
            button?.isHighlighted = false
        }
        self.activityIndicator.stopAnimating()
    }
```

```
    // MARK: - view setup

    override func viewDidLoad() {
        super.viewDidLoad()
        self.activityIndicator.stopAnimating()

        // Do any additional setup after loading the view.

        coordinatedPipeline = createFuturePublisher(button: self.step1_button) ⑦
            .flatMap { flatMapInValue -> AnyPublisher<Bool, Error> in
            let step2_1 = self.createFuturePublisher(button: self.step2_1_button)
            let step2_2 = self.createFuturePublisher(button: self.step2_2_button)
            let step2_3 = self.createFuturePublisher(button: self.step2_3_button)
            return Publishers.Zip3(step2_1, step2_2, step2_3)
                .map { _ -> Bool in
                    return true
                }
                .eraseToAnyPublisher()
            }
        .flatMap { _ in
            return self.createFuturePublisher(button: self.step3_button)
        }
        .flatMap { _ in
            return self.createFuturePublisher(button: self.step4_button)
        }
        .eraseToAnyPublisher()
    }
}
```

① `runItAll` coordinates the operation of this little workflow, starting with checking to see if one is currently running. If defined, it calls the cancel on the existing subscriber.

② `resetAllSteps` iterates through all the existing buttons used represent the progress of this workflow, and resets them to gray and unhighlighted to reflect an initial state. It also verifies that the activity indicator is not currently animated.

③ Then we get things started, first with activating the animation on the activity indicator.

④ Creating the subscriber with sink and storing the reference initiates the workflow. The publisher to which it is subscribing is setup outside this function, allowing it to be re-used multiple times. The print operator in the pipeline is for debugging, to show console output of when the pipeline is triggered.

⑤ Each step is represented by the invocation of a Future publisher, followed immediately by pipeline elements to switch to the main thread and then update a UIButton's background to show the step has completed. This is encapsulated in a `createFuturePublisher` call, using eraseToAnyPublisher to simplify the type being returned.

⑥ The map operator is used to create this specific side effect of updating the a UIbutton to show the step has been completed.

⑦ The creation of the overall pipeline and its structure of serial and parallel tasks is created from

the combination of calls to `createFuturePublisher` along with the operators flatMap and zip.

Error Handling

The examples above expected that the subscriber would handle the error conditions, if they occured. However, you are not always able to control the subscriber - as might be the case if you're using SwiftUI view properties as the subscriber, and you're providing the publisher. In these cases, you need to build your pipeline so that the output types match the subscriber types.

For example, if you are working with SwiftUI and the you want to use assign to set the `isEnabled` property on a button, the subscriber will have a few requirements:

1. the subcriber should match the type output of `<Bool>`, `<Never>`
2. the subscriber should be called on the main thread

With a publisher that can throw an error (such as URLSession.dataTaskPublisher), you need to construct a pipeline to convert the output type, but also handle the error within the pipeline to match a failure type of `<Never>`.

How you handle the errors within a pipeline is very dependent on how the pipeline is working. If the pipeline is set up to return a single result and terminate, continue to Using catch to handle errors in a one-shot pipeline. If the pipeline is set up to continually update, the error handling needs to be a little more complex. Jump ahead to Using flatMap with catch to handle errors.

Verifying a failure hasn't happened using assertNoFailure

Goal

- Verify no error has occured within a pipeline

References

- assertNoFailure

See also

- << link to other patterns>>

Code and explanation

> Useful in testing invariants in pipelines, the assertNoFailure operator also converts the failure type to `<Never>`. The operator will cause the application to terminate (and tests to crash to a debugger) if the assertion is triggered.

This is useful for verifying the invariant of having dealt with an error. If you are sure you handled the errors and need to map a pipeline which technically can generate a failure type of `<Error>` to a subscriber that requires a failure type of `<Never>`.

It is far more likely that you want to handle the error with and not have the application terminate. Look forward to Using catch to handle errors in a one-shot pipeline and Using flatMap with catch to handle errors for patterns of how to provide logic to handle errors in a pipeline.

Using catch to handle errors in a one-shot pipeline

Goal

- If you need to handle a failure within a pipeline, for example before using the `assign` operator or another operator that requires the failure type to be `<Never>`, you can use `catch` to provide the appropriate logic.

References

- catch
- Just

See also

- Retrying in the event of a temporary failure
- Using flatMap with catch to handle errors
- Requesting data from an alternate URL when the network is constrained

Code and explanation

catch handles errors by replacing the upstream publisher with another publisher that you provide as a return in a closure.

Be aware that this effectively terminates the earlier portion of the pipeline. If you're using a one-shot publisher (one that doesn't create more than a single event), then this is fine.

For example, URLSession.dataTaskPublisher is a one-shot publisher and you might use catch with it to ensure that you get a response, returning a placeholder in the event of an error. Extending our previous example to provide a default response:

```
struct IPInfo: Codable {
    // matching the data structure returned from ip.jsontest.com
    var ip: String
}
let myURL = URL(string: "http://ip.jsontest.com")
// NOTE(heckj): you'll need to enable insecure downloads in your Info.plist for this
example
// since the URL scheme is 'http'

let remoteDataPublisher = URLSession.shared.dataTaskPublisher(for: myURL!)
    // the dataTaskPublisher output combination is (data: Data, response: URLResponse)
    .map({ (inputTuple) -> Data in
        return inputTuple.data
    })
    .decode(type: IPInfo.self, decoder: JSONDecoder()) ①
    .catch { err in ②
        return Publishers.Just(IPInfo(ip: "8.8.8.8"))③
    }
    .eraseToAnyPublisher()
```

① Often, a catch operator will be placed after several operators that could fail, in order to provide a fallback or placeholder in the event that any of the possible previous operations failed.

② When using catch, you get the error type in and can inspect it to choose how you provide a response.

③ The Just publisher is frequently used to either start another one-shot pipeline or to directly provide a placeholder response in the event of failure.

A possible problem with this technique is that the if the original publisher generates more values to which you wish to react, the original pipeline has been ended. If you are creating a pipeline that reacts to a Published property, then after any failed value that activates the catch operator, the pipeline will cease to react further. See catch for more illustration and examples of how this works.

If you want to continue to respond to errors and handle them, see Using flatMap with catch to handle errors for an example of how to do that using flatMap.

Retrying in the event of a temporary failure

Goal

- The retry operator can be included in a pipeline to retry a subscription when a `.failure` completion occurs.

References

- catch
- retry
- delay
- tryMap

See also

- Using catch to handle errors in a one-shot pipeline
- Using flatMap with catch to handle errors

Code and explanation

When requesting data from a dataTaskPublisher, there is the possibility that the request may fail, and instead of data you may receive a `.failure` completion with an error. When it fails, the retry operator will let you retry that same request for a set number of attempts that you specify. When the publisher doesn't send a failure - sending it's result data type - the retry operator passes through the resulting values. It only reacts within a combine pipeline when a `.failure` completion is sent.

When it does sense a `.failure` completion, the way it retries is by re-doing the subscription to the operator or publisher to which it was chained.

The retry operator is commonly desired when attempting to request network resources with an unstable connection, or other situations where the answer might change if the request happens again after a short period of time. If the number of retries specified all fail, then the `.failure` completion is passed down to the subscriber of this operator.

In our example below, we are using retry in combination with a delay operator. The delay operator puts a small random delay in how long the request takes. This spaces out the retry attempts, so that the retries don't all happen in quick succession.

This example also includes the use of the tryMap operator to more fully inspect any URLResponse returned from the dataTaskPublisher. Any response from the server is encapsaulted by URLSession, and passed forward as a valid response. Because of this, URLSession doesn't treat a 404 Not Found http response as an error response, nor any 50x error codes. Using tryMap lets us inspect the response code that was sent, and verify that it was a 200 response code. In this example, if the response code is anything but a 200 response, it throws an exception - which in turn causes the tryMap operator to pass down a `.failure` completion rather than data. This example sets the tryMap **after** the retry operator so that retry will only re-attempt the request when the site didn't respond.

```
let remoteDataPublisher = urlSession.dataTaskPublisher(for: self.mockURL!)
    .delay(for: DispatchQueue.SchedulerTimeType.Stride(integerLiteral: Int.random(in:
1..<5)), scheduler: backgroundQueue) ①
    .retry(3) ②
    .tryMap { data, response -> Data in ③
        guard let httpResponse = response as? HTTPURLResponse,
            httpResponse.statusCode == 200 else {
                throw TestFailureCondition.invalidServerResponse
        }
        return data
    }
    .decode(type: PostmanEchoTimeStampCheckResponse.self, decoder: JSONDecoder())
    .subscribe(on: backgroundQueue)
    .eraseToAnyPublisher()
```

① The delay operator will hold the results flowing through the pipeline for a short duration, in this case for a random selection of 1 to 5 seconds. By adding delay here in the pipeline, it will always occur, even if the original request is successful.

② Retry is specified as trying 3 times.

③ tryMap is being used to inspect the data result from dataTaskPublisher and return a `.failure` completion if the response from the server is valid, but not a 200 HTTP response code.

When using the retry operator with URLSession.dataTaskPublisher, verify that the URL you are requesting isn't going to have negative side effects if requested repeatedly or with a retry. Ideally such requests are be expected to be idempotent. If they are not, the retry operator may make multiple requests, with very unexpected side effects.

Using flatMap with catch to handle errors

Goal

- The `flatMap` operator can be used with `catch` to continue to handle errors on new published values.

References

- flatMap
- Just
- catch

See also

- Using catch to handle errors in a one-shot pipeline
- Retrying in the event of a temporary failure

Code and explanation

The `flatMap` operator is the operator to use in handling errors on a continual flow of events.

You provide a closure to flatMap that can read in the value that was provided, and creates a one-shot closure that does the possibly failing work. An example of this is requesting data from a network and then decoding the returned data. You can include a catch operator to capture any errors and provide any appropriate value.

This is a perfect mechanism for when you want to maintain updates up an upstream publisher, as it creates one-shot publisher or short pipelines that send a single value and then complete for every incoming value. The completion from the created one-shot publishers terminates in the flatMap and isn't passed to downstream subscribers.

An example of this with a dataTaskPublisher:

```
let remoteDataPublisher = Just(self.testURL!) ①
    .flatMap { url in ②
        URLSession.shared.dataTaskPublisher(for: url) ③
        .tryMap { data, response -> Data in ④
            guard let httpResponse = response as? HTTPURLResponse,
                httpResponse.statusCode == 200 else {
                    throw TestFailureCondition.invalidServerResponse
            }
            return data
        }
        .decode(type: PostmanEchoTimeStampCheckResponse.self, decoder: JSONDecoder())
⑤
        .catch {_ in ⑥
            return Just(PostmanEchoTimeStampCheckResponse(valid: false))
        }
    }
    .subscribe(on: self.myBackgroundQueue!)
    .eraseToAnyPublisher()
```

① Just starts this publisher as an example by passing in a URL.

② flatMap takes the URL as input and the closure goes on to create a one-shot publisher chain.

③ dataTaskPublisher uses the input url

④ which flows to tryMap to parse for additional errors

⑤ and finally decode to attempt to refine the returned data into a local type

⑥ if any of these have failed, catch will convert the error into a placeholder sample, in this case an object with a preset `valid = false` property.

Requesting data from an alternate URL when the network is constrained

Goal

- From Apple's WWDC 19 presentation Advances in Networking, Part 1, a sample pattern was provided using `tryCatch` and `tryMap` operators to react to the specific error of having the network be constrained.

References

- URLSession.dataTaskPublisher
- tryCatch
- tryMap

See also

- Using catch to handle errors in a one-shot pipeline
- Retrying in the event of a temporary failure

Code and explanation

This sample is originally from the WWDC session. The API and example is evolving with the beta releases of Combine since that presentation. `tryCatch` was missing in the beta2 release, and has returned in beta3.

```
// Generalized Publisher for Adaptive URL Loading
func adaptiveLoader(regularURL: URL, lowDataURL: URL) -> AnyPublisher<Data, Error> {
    var request = URLRequest(url: regularURL) ①
    request.allowsConstrainedNetworkAccess = false ②
    return URLSession.shared.dataTaskPublisher(for: request) ③
        .tryCatch { error -> URLSession.DataTaskPublisher in ④
            guard error.networkUnavailableReason == .constrained else {
               throw error
            }
            return URLSession.shared.dataTaskPublisher(for: lowDataURL) ⑤
        .tryMap { data, response -> Data in
            guard let httpResponse = response as? HTTPUrlResponse, ⑥
                   httpResponse.status_code == 200 else {
                       throw MyNetworkingError.invalidServerResponse
            }
            return data
}
.eraseToAnyPublisher() ⑦
```

This example, from Apple's WWDC, provides a function that takes two URLs - a primary and a fallback. It returns a publisher that will request data and fall back requesting a secondary URL when the network is constrained.

① The request starts with an attempt requesting data.

② Setting `request.allowsConstrainedNetworkAccess` will cause the dataTaskPublisher to error if the

network is constrained.

③ Invoke the dataTaskPublisher to make the request.

④ tryCatch is used to capture the immediate error condition and check for a specific error (the constrained network).

⑤ If it finds an error, it creates a new one-shot publisher with the fall-back URL.

⑥ The resulting publisher can still fail, and tryMap can map this a failure by throwing an error on HTTP response codes that map to error conditions

⑦ eraseToAnyPublisher will do type erasure on the chain of operators so the resulting signature of the adaptiveLoader function is of type `AnyPublisher<Data, Error>`

In the sample, if the error returned from the original request wasn't an issue of the network being constrained, it passes on the .failure completion down the pipeline. If the error is that the network is constrained, then the tryCatch operator creates a new request to an alternate URL.

UIKit (or AppKit) Integration

Declarative UI updates from user input

Goal

- Querying a web based API and returning the data to be displayed in your UI

References

- The Xcode project ViewController with this code is in the github project at `UIKit-Combine/GithubViewController.swift`
- Publishers: Published, URLSession.dataTaskPublisher,
- Operators: map, switchToLatest, receive, throttle, removeDuplicates
- Subscribers: assign

See also

- Using flatMap with catch to handle errors
- Using catch to handle errors in a one-shot pipeline
- Stricter request processing with dataTaskPublisher

Code and explanation

One of the primary benefits of a framework like Combine is setting up a declarative structure that defines how an interface will update to user input.

A pattern for integrating UIKit is setting up a variable which will hold a reference to the updated state, and then linking that with existing UIKit or AppKit controls within IBAction.

The sample here is a portion of the code at in a larger ViewController implementation.

This example overlaps with the next pattern Cascading UI updates including a network request, which builds upon the initial publisher.

UIKit-Combine/GithubAPI.swift

```
import UIKit
import Combine

class ViewController: UIViewController {

    @IBOutlet weak var github_id_entry: UITextField! ①

    var usernameSubscriber: AnyCancellable?

    // username from the github_id_entry field, updated via IBAction
    @Published var username: String = "" ②

    // github user retrieved from the API publisher. As it's updated, it
    // is "wired" to update UI elements
    @Published private var githubUserData: [GithubAPIUser] = []

    // MARK - Actions

    @IBAction func githubIdChanged(_ sender: UITextField) {
        username = sender.text ?? "" ③
        print("Set username to ", username)
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.

        usernameSubscriber = $username ④
            .throttle(for: 0.5, scheduler: myBackgroundQueue, latest: true) ⑤
            // ^^ scheduler myBackGroundQueue publishes resulting elements
            // into that queue, resulting on this processing moving off the
            // main runloop.
            .removeDuplicates() ⑥
            .print("username pipeline: ") // debugging output for pipeline
            .map { username -> AnyPublisher<[GithubAPIUser], Never> in ⑦
                return GithubAPI.retrieveGithubUser(username: username)
            }
            // ^^ type returned in the pipeline is a Publisher, so we use
            // switchToLatest to flatten the values out of that
            // pipeline to return down the chain, rather than returning a
            // publisher down the pipeline.
            .switchToLatest() ⑧
            // using a sink to get the results from the API search lets us
            // get not only the user, but also any errors attempting to get it.
            .receive(on: RunLoop.main)
            .assign(to: \.githubUserData, on: self) ⑨
```

① The UITextField is the interface element which is driving the updates from user interaction.

② We defined a Published property to both hold the updates. Because its a Published property, it provides a publisher reference that we can use to attach additional Combine pipelines to update other variables or elements of the interface.

③ We set the variable `username` from within an IBAction, which in turn triggers a data flow if the publisher `$username` has any subscribers.

④ We in turn set up a subscriber on the publisher `$username` that does further actions. In this case the overall flow retrives an instance of a GithubAPIUser from Github's REST API.

⑤ The throttle is there to keep from triggering a network request on every request. The throttle keeps it to a maximum of 1 request every half-second.

⑥ removeDuplicates is there to collapse events from the changing username so that API requests aren't made on rapidly changing values. The removeDuplicates prevents redundant requests from being made, should the user edit and the return the previous value.

⑦ map is used similiarly to flatMap in error handling here, returning an instance of a publisher. The API object returns a publisher, which this map is invoking. This doesn't return the value from the call, but the calling publisher itself.

⑧ switchToLatest operator takes the instance of the publisher that is the element passed down the pipeline, and pulls out the data to push the elements further down the pipeline. switchToLatest resolves that publisher into a value and passes that value down the pipeline, in this case an instance of `[GithubAPIUser]`.

⑨ And assign at the end up the pipeline is the subscriber, which assigns the value into another variable

Continue on to Cascading UI updates including a network request to expand this into multiple cascading updates of various UI elements.

Cascading UI updates including a network request

Goal

- Have multiple UI elements update triggered by an upstream subscriber

References

- The ViewController with this code is in the github project at UIKit-Combine/GithubViewController.swift
- The GithubAPI is in the github project at UIKit-Combine/GithubAPI.swift
- Publishers: Published, URLSession.dataTaskPublisher, Just, Empty
- Operators: decode, catch, map, tryMap, switchToLatest, filter, handleEvents, subscribe, receive, throttle, removeDuplicates
- Subscribers: sink, assign

See also

- Using flatMap with catch to handle errors
- Using catch to handle errors in a one-shot pipeline
- Stricter request processing with dataTaskPublisher

Code and explanation

The example provided expands on a publisher updating from Declarative UI updates from user input, adding additional Combine pipelines to update multiple UI elements.

The general pattern of this view starts with a textfield that accepts user input:

1. We using an IBAction to update the Published username variable.
2. We have a subscriber (`usernameSubscriber`) attached `$username` publisher reference, which attempts to retrieve the GitHub user from the API. The resulting variable `githubUserData` (also Published) is a list of GitHub user objects. Even though we only expect a single value here, we use a list because we can conveniently return an empty list on failure scenarios: unable to access the API or the username isn't registered at GitHub.
3. We have a "secondary" subscriber `apiNetworkActivitySubscriber` which another publisher from the GithubAPI object that provides values when the GithubAPI object starts or finishes making network requests.
4. We have a another subscriber `repositoryCountSubscriber` attached to `$githubUserData` that pulls the repository count off the github user data object and assigns it as the count to be displayed.
5. We have a final subscriber `avatarViewSubscriber` attached to `$githubUserData` that attempts to retrieve the image associated with the user's avatar for display.

The empty list is useful to return because when a username is provided that doesn't resolve, we want to explicitly remove any avatar image that was previously displayed. To do this, we need the pipelines to fully resolve to some value, so that further pipelines are triggered and the relevant UI interfaces updated.

The subscribers (created with assign and sink) are stored as AnyCancellable variables on the

ViewController instance. Because they are defined on the class instance, the Swift compiler creates initializers and deinitializers, which will cancel and clean up the publishers when the class is torn down.

A number of developers comfortable with RxSwift are using a "CancelBag" object to collect cancellable references, and cancel the pipelines on tear down. An example of this can be seen at https://github.com/tailec/CombineExamples/blob/master/CombineExamples/Shared/CancellableBag.swift

The pipelines have been explicitly configured to work on a background queue using the subscribe operator. Without that additional configured, the pipelines would be invoked and run on the main runloop since they were invoked from the UI, which causes a noticable slow-down in responsiveness in the simulator. Likewise when the resulting pipelines assign or update UI elements, the receive operator is used to transfer that work back onto the main runloop.

If you want to have the UI continuously updated from changes propogating through Published properties, make sure that any configured pipelines have a <Never> failure type. This is required for the assign operator. But more importantly, it's a source of bugs when using a sink operator. If the pipeline from a Published variable terminates in a sink that accepts an Error failure type, the sink will send a termination signal if an error occures, which stops the pipeline from further processing even when the variable is updated.

UIKit-Combine/GithubAPI.swift

```
import Founation
import Combine

enum APIFailureCondition: Error {
    case invalidServerResponse
}

struct GithubAPIUser: Decodable { ①
    // A very *small* subset of the content available about
    //  a github API user for example:
    // https://api.github.com/users/heckj
    let login: String
    let public_repos: Int
    let avatar_url: String
}

struct GithubAPI { ②
    // NOTE(heckj): I've also seen this kind of API access
    // object set up with with a class and static methods on the class.
    // I don't know that there's a specific benefit to make this a value
    // type/struct with a function on it.

    /// externally accessible publsher that indicates that network activity is
happening in the API proxy
```

```
    static let networkActivityPublisher = PassthroughSubject<Bool, Never>() ③

    /// creates a one-shot publisher that provides a GithubAPI User
    /// object as the end result. This method was specifically designed to
    /// return a list of 1 object, as opposed to the object itself to make
    /// it easier to distinguish a "no user" result (empty list)
    /// representation that could be dealt with more easily in a Combine
    /// pipeline than an optional value. The expected return types is a
    /// Publisher that returns either an empty list, or a list of one
    /// GithubAPUser, and with a failure return type of Never, so it's
    /// suitable for recurring pipeline updates working with a @Published
    /// data source.
    /// - Parameter username: username to be retrieved from the Github API
    static func retrieveGithubUser(username: String) -> AnyPublisher<[GithubAPIUser],
Never> { ④

        if username.count < 3 { ⑤
            return Just([]).eraseToAnyPublisher()
            // return Publishers.Empty<GithubAPIUser, Never>()
            //    .eraseToAnyPublisher()
        }
        let assembledURL = String("https://api.github.com/users/\(username)")
        let publisher = URLSession.shared.dataTaskPublisher(for: URL(string:
assembledURL)!)
            .handleEvents(receiveSubscription: { _ in ⑥
                networkActivityPublisher.send(true)
            }, receiveCompletion: { _ in
                networkActivityPublisher.send(false)
            }, receiveCancel: {
                networkActivityPublisher.send(false)
            })
            .tryMap { data, response -> Data in ⑦
                guard let httpResponse = response as? HTTPURLResponse,
                    httpResponse.statusCode == 200 else {
                        throw APIFailureCondition.invalidServerResponse
                }
                return data
            }
            .decode(type: GithubAPIUser.self, decoder: JSONDecoder()) ⑧
            .map {
                [$0] ⑨
            }
            .catch { err in ⑩
                // return Publishers.Empty<GithubAPIUser, Never>()
                // ^^ when I originally wrote this method, I was returning
                // a GithubAPIUser? optional, and then a GithubAPIUser without
                // optional. I ended up converting this to return an empty
                // list as the "error output replacement" so that I could
                // represent that the current value requested didn't *have* a
                // correct github API response. When I was returing a single
                // specific type, using Publishers.Empty was a good way to do a
```

```
                // "no data on failure" error capture scenario.
                return Just([])
            }
            .eraseToAnyPublisher() ⑪
            return publisher
    }
}
```

① The decodable struct created here is a subset of what's returned from the GitHub API. Any pieces not defined in the struct are simply ignored when processed by the decode operator.

② The code to interact with the GitHub API was broken out into its own object, which I would normally have in a separate file. The functions on the API struct return publishers, and are then mixed and merged with other pipelines in the ViewController.

③ This struct also exposes a publisher using passthroughSubject that have set up to trigger Boolean values when it is actively making network requests.

④ I first created the pipelines to return an optional GithubAPIUser instance, but found that there wasn't a convenient way to propogate "nil" or empty objects on failure conditions. The code was then recreated to return a list, even though only a single instance was ever expected, to conveniently represent an "empty" object. This was important for the use case of wanting to erase existing values in following pipelines reacting to the GithubAPIUser object "disappearing" - removing the repository count and avatar images in this case.

⑤ The logic here is simply to prevent extraneous network requests, returning an empty result if the username being requested has less than 3 characters. The commented out code is a bit of legacy from when I wanted to return nothing instead of an empty list.

⑥ the handleEvents operator here is how we are triggering updates for the network activity publisher. We define closures that trigger on subscription and finalization (both completion and cancel) that invoke `send()` on the passthroughSubject. This is an example of how we can provide metadata about a pipeline's operation as a separate publisher.

⑦ tryMap adds additional checking on the API response from github to convert correct responses from the API that aren't valid User instances into a pipeline failure condition.

⑧ decode takes the Data from the response and decodes it into a single instance of `GithubAPIUser`

⑨ map is used to take the single instance and convert it into a list of 1 item, changing the type to a list of `GithubAPIUser`: `[GithubAPIUser]`.

⑩ catch operator captures the error conditions within this pipeline, and returns an empty list on failure while also converting the failure type to `Never`.

⑪ eraseToAnyPublisher collapses the complex types of all the chained operators and exposes the whole pipeline as an instance of `AnyPublisher`.

UIKit-Combine/GithubViewController.swift

```
import UIKit
import Combine

class ViewController: UIViewController {
```

```
    @IBOutlet weak var github_id_entry: UITextField!
    @IBOutlet weak var activityIndicator: UIActivityIndicatorView!
    @IBOutlet weak var repositoryCountLabel: UILabel!
    @IBOutlet weak var githubAvatarImageView: UIImageView!

    var repositoryCountSubscriber: AnyCancellable?
    var avatarViewSubscriber: AnyCancellable?
    var usernameSubscriber: AnyCancellable?
    var headingSubscriber: AnyCancellable?
    var apiNetworkActivitySubscriber: AnyCancellable?

    // username from the github_id_entry field, updated via IBAction
    @Published var username: String = ""

    // github user retrieved from the API publisher. As it's updated, it
    // is "wired" to update UI elements
    @Published private var githubUserData: [GithubAPIUser] = []

    // publisher reference for this is $username, of type <String, Never>
    var myBackgroundQueue: DispatchQueue = DispatchQueue(label:
"viewControllerBackgroundQueue")
    let coreLocationProxy = LocationHeadingProxy()

    // MARK - Actions

    @IBAction func githubIdChanged(_ sender: UITextField) {
        username = sender.text ?? ""
        print("Set username to ", username)
    }

    // MARK - lifecycle methods

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.

        let apiActivitySub = GithubAPI.networkActivityPublisher ①
        .receive(on: RunLoop.main)
            .sink { doingSomethingNow in
                if (doingSomethingNow) {
                    self.activityIndicator.startAnimating()
                } else {
                    self.activityIndicator.stopAnimating()
                }
        }
        apiNetworkActivitySubscriber = AnyCancellable(apiActivitySub)

        usernameSubscriber = $username ②
            .throttle(for: 0.5, scheduler: myBackgroundQueue, latest: true)
            // ^^ scheduler myBackGroundQueue publishes resulting elements
            // into that queue, resulting on this processing moving off the
```

```
            // main runloop.
            .removeDuplicates()
            .print("username pipeline: ") // debugging output for pipeline
            .map { username -> AnyPublisher<[GithubAPIUser], Never> in
                return GithubAPI.retrieveGithubUser(username: username)
            }
            // ^^ type returned in the pipeline is a Publisher, so we use
            // switchToLatest to flatten the values out of that
            // pipeline to return down the chain, rather than returning a
            // publisher down the pipeline.
            .switchToLatest()
            // using a sink to get the results from the API search lets us
            // get not only the user, but also any errors attempting to get it.
            .receive(on: RunLoop.main)
            .assign(to: \.githubUserData, on: self)

        // using .assign() on the other hand (which returns an
        // AnyCancellable) *DOES* require a Failure type of <Never>
        repositoryCountSubscriber = $githubUserData ③
            .print("github user data: ")
            .map { userData -> String in
                if let firstUser = userData.first {
                    return String(firstUser.public_repos)
                }
                return "unknown"
            }
            .receive(on: RunLoop.main)
            .assign(to: \.text, on: repositoryCountLabel)

        let avatarViewSub = $githubUserData ④
            // When I first wrote this publisher pipeline, the type I was
            // aiming for was <GithubAPIUser?, Never>, where the value was an
            // optional. The commented out .filter below was to prevent a `nil` //
GithubAPIUser object from propogating further and attempting to
            // invoke the dataTaskPublisher which retrieves the avatar image.
            //
            // When I updated the type to be non-optional (<GithubAPIUser?,
            // Never>) the filter expression was no longer needed, but possibly
            // interesting.
            // .filter({ possibleUser -> Bool in
            //     possibleUser != nil
            // })
            // .print("avatar image for user") // debugging output
            .map { userData -> AnyPublisher<UIImage, Never> in
                guard let firstUser = userData.first else {
                    // my placeholder data being returned below is an empty
                    // UIImage() instance, which simply clears the display.
                    // Your use case may be better served with an explicit
                    // placeholder image in the event of this error condition.
                    return Just(UIImage()).eraseToAnyPublisher()
                }
```

```
                return URLSession.shared.dataTaskPublisher(for: URL(string: firstUser
.avatar_url)!)
                    // ^^ this hands back (Data, response) objects
                    .handleEvents(receiveSubscription: { _ in
                        DispatchQueue.main.async {
                            self.activityIndicator.startAnimating()
                        }
                    }, receiveCompletion: { _ in
                        DispatchQueue.main.async {
                            self.activityIndicator.stopAnimating()
                        }
                    }, receiveCancel: {
                        DispatchQueue.main.async {
                            self.activityIndicator.stopAnimating()
                        }
                    })
                    .map { $0.data }
                    // ^^ pare down to just the Data object
                    .map { UIImage(data: $0)!}
                    // ^^ convert Data into a UIImage with its initializer
                    .subscribe(on: self.myBackgroundQueue)
                    // ^^ do this work on a background Queue so we don't screw
                    // with the UI responsiveness
                    .catch { err in
                        return Just(UIImage())
                    }
                    // ^^ deal the failure scenario and return my "replacement"
                    // image for when an avatar image either isn't available or
                    // fails somewhere in the pipeline here.
                    .eraseToAnyPublisher()
                    // ^^ match the return type here to the return type defined
                    // in the .map() wrapping this because otherwise the return
                    // type would be terribly complex nested set of generics.
            }
            .switchToLatest()
            // ^^ Take the returned publisher that's been passed down the chain
            // and "subscribe it out" to the value within in, and then pass
            // that further down.
            .subscribe(on: myBackgroundQueue)
            // ^^ do the above processing as well on a background Queue rather
            // than potentially impacting the UI responsiveness
            .receive(on: RunLoop.main)
            // ^^ and then switch to receive and process the data on the main
            // queue since we're messin with the UI
            .map { image -> UIImage? in
                image
            }
            // ^^ this converts from the type UIImage to the type UIImage?
            // which is key to making it work correctly with the .assign()
            // operator, which must map the type *exactly*
            .assign(to: \.image, on: self.githubAvatarImageView)
```

```
        // convert the .sink to an `AnyCancellable` object that we have
        // referenced from the implied initializers
        avatarViewSubscriber = AnyCancellable(avatarViewSub)

        // KVO publisher of UIKit interface element
        let _ = repositoryCountLabel.publisher(for: \.text) ⑤
            .sink { someValue in
                print("repositoryCountLabel Updated to \(String(describing: someValue
))")
        }
    }

}
```

① We add a subscriber to our previous controller from that connects notifications of activity from the GithubAPI object to our activity indicator.

② Where the username is updated from the IBAction (from our earlier example Declarative UI updates from user input) we have the subscriber make the network request and put the results in a new variable (also published) on our ViewController.

③ The first of two subscribers on the publisher `$githubUserData`, this pipeline extracts the count of repositories and updates the UI label instance. There is a bit of logic in the middle of the pipeline to return the string "unknown" when the list is empty.

④ The second subscriber to the publisher `$githubUserData`, this triggers a follow on network request to request the image data for the github avatar. This is a more complex pipeline, extracting the data from the githubUser, assembling a URL, and then requesting it. As this code is in the ViewController, we can also use handleEvents operator to trigger updates to the activityIndicator in our view. We use subscribe to make the requests on a background queue, and later receive the results back onto the main thread to update the UI elements. The catch and failure handling returns an empty `UIImage` instance in the event of failure.

⑤ A final subscriber that doesn't do anything is attached to the UILabel itself. Any Key-Value Observable object from Foundation can also produce a publisher. In this example, we attach a publisher that triggers a print statement that the UI element was updated.

While we could simply attach pipelines to UI elements as we're updating them, it more closely couples interactions to the actual UI elements themselves. While easy and direct, it is often a good idea to make explicit state and updates to seperate out actions and data for debugging and understandability. In the example above, we use two published properties to hold the state associated with the current view. One of which is updated by an IBAction, and the second updated declaratively using a Combine publisher pipeline. All other UI elements are updated publishers hanging from those properties getting updated.

Merging multiple pipelines to update UI elements

Goal

- Watch and react to multiple UI elements publishing values, and updating the interface based on the combination of values updated.

References

- The ViewController with this code is in the github project at UIKit-Combine/FormViewController.swift
- Publishers: Published,
- Operators: combineLatest, map, receive,
- Subscribers: assign

See also

- Declarative UI updates from user input

Code and explanation

This example intentionally mimics a lot of web form style validation scenarios, but within UIKit and using Comine.

A view controller is set up with multiple elements to declaratively update. The view controller hosts 3 primary text input fields: * value1 * value2 * value2_repeat a button to submit the combined values, and two labels to provide feedback messages.

The rules of these update that are implemented: * the entry in value1 has to be at least 3 characters * the entry in value2 has to be at least 5 characters * the entry in value2_repeat has to be the same as value2

If any of these rules aren't met, then we want the submit button to be disabled and relevant messages displayed explaining what needs to be done.

This is achieved by setting up a cascade of pipelines that link and merge together.

- At the base, there are Published property matching each of the user input fields. combineLatest is used to take the continually published updates from the value2 properties and merge them into a single pipeline. A map operator enforces the rule about characters required and that the values need to be the same. If the values don't match the required output, we pass a nil value down the pipeline.
- Another validation pipeline is set up for value1, just using a map operator to validate the value, or return nil.
- The logic within the map operators doing the validation is also used to update the label messages in the user interface.
- A final pipeline uses combineLatest to merge the two validation pipelines into a single pipeline. A subscriber is attached to this combined pipeline to determine if the submission button should be enabled.

The example below shows the various pieces all connected.

UIKit-Combine/FormViewController.swift

```
import UIKit
import Combine

class FormViewController: UIViewController {

    @IBOutlet weak var value1_input: UITextField!
    @IBOutlet weak var value2_input: UITextField!
    @IBOutlet weak var value2_repeat_input: UITextField!
    @IBOutlet weak var submission_button: UIButton!
    @IBOutlet weak var value1_message_label: UILabel!
    @IBOutlet weak var value2_message_label: UILabel!

    @IBAction func value1_updated(_ sender: UITextField) { ①
        value1 = sender.text ?? ""
    }
    @IBAction func value2_updated(_ sender: UITextField) {
        value2 = sender.text ?? ""
    }
    @IBAction func value2_repeat_updated(_ sender: UITextField) {
        value2_repeat = sender.text ?? ""
    }

    @Published var value1: String = ""
    @Published var value2: String = ""
    @Published var value2_repeat: String = ""

    var validatedValue1: AnyPublisher<String?, Never> { ②
        return $value1.map { value1 in
            guard value1.count > 2 else {
                DispatchQueue.main.async { ③
                    self.value1_message_label.text = "minimum of 3 characters
required"
                }
                return nil
            }
            DispatchQueue.main.async {
                self.value1_message_label.text = ""
            }
            return value1
        }.eraseToAnyPublisher()
    }

    var validatedValue2: AnyPublisher<String?, Never> { ④
        return Publishers.CombineLatest($value2, $value2_repeat)
            .receive(on: RunLoop.main) ⑤
            .map { value2, value2_repeat in
                guard value2_repeat == value2, value2.count > 4 else {
                    self.value2_message_label.text = "values must match and have at
least 5 characters"
```

```
                    return nil
                }
                self.value2_message_label.text = ""
                return value2
            }.eraseToAnyPublisher()
    }

    var readyToSubmit: AnyPublisher<(String, String)?, Never> { ⑥
        return Publishers.CombineLatest(validatedValue2, validatedValue1)
            .map { value2, value1 in
                guard let realValue2 = value2, let realValue1 = value1 else {
                    return nil
                }
                return (realValue2, realValue1)
            }
            .eraseToAnyPublisher()
    }

    private var cancellableSet: Set<AnyCancellable> = [] ⑦

    override func viewDidLoad() {
        super.viewDidLoad()

        self.readyToSubmit
            .map { $0 != nil } ⑧
            .receive(on: RunLoop.main)
            .assign(to: \.isEnabled, on: submission_button)
            .store(in: &cancellableSet) ⑨
    }
}
```

① The start of this code follows the same patterns laid out in Declarative UI updates from user input. IBAction messages are used to update the Published properties, triggering updates to any subscribers attached.

② The first of the validation pipelines uses a map operator to take the string value intput and convert it to nil if it doesn't match the validation rules. This is also converting the output type from the published property of `<String>` to the optional `<String?>`. The same logic is also used to trigger updates to the messages label to provide information about what is required.

③ Since we are updating user interface elements, we explicitly make those updates wrapped in `DispatchQueue.main.async` to invoke on the main thread.

④ combineLatest takes two publishers and merges them into a single pipeline with an output type that is the combined values of each of the upstream publishers. In this case, the output type is a tuple of `(<String>, <String>)`.

⑤ Rather than use `DispatchQueue.main.async`, we can use the receive operator to explicitly run the next operator on the main thread, since it will be doing UI updates.

⑥ The two validation pipelines are combined with combineLatest, and the output of those checked and merged into a single tuple output.

⑦ We could store the assignment pipeline as an AnyCancellable? reference to map it to the life of the viewcontroller, but another option is to create something to collect all the cancellable references. This starts as an empty set, and any sinks or assignment subscribers can be added to it to keep a reference to them so that they operate over the full lifetime of the view controller.

⑧ If any of the values are nil, the map operator returns nil down the pipeline. Checking against a nil value provides the boolean used to enable (or disable) the submission button.

⑨ the `store` method is available on the Cancellable protocol, which is explicitly set up to support saving off references that can be used to cancel a pipeline.

Creating a repeating publisher by wrapping a delegate based API

Goal

- To use one of the Apple delegate APIs to provide values to be used in a Combine pipeline.

References

- passthroughSubject
- currentValueSubject

See also

- Wrapping an asynchronous call with a Future to create a one-shot publisher
- passthroughSubject
- delay

Code and explanation

> Where a Future publisher is great for wrapping existing code to make a single request, it doesn't serve as well to make a publisher that produces lengthy, or potentially unbounded, amount of output.

Apple's UIKit and AppKit APIs have tended to have a object/delegate pattern, where you can opt in to receiving any number of different callbacks (often with data). One such example of that is included within the CoreLocation library, which offers a number of different data sources.

If you want to consume data provided by one of these kinds of APIs within a pipeline, you can wrap the object and use passthroughSubject to expose a publisher. The sample code belows shows an example of wrapping CoreLocation's CLManager object and consuming the data from it through a UIKit view controller.

UIKit-Combine/LocationHeadingProxy.swift

```
import Foundation
import Combine
import CoreLocation

final class LocationHeadingProxy: NSObject, CLLocationManagerDelegate {

    let mgr: CLLocationManager ①
    private let headingPublisher: PassthroughSubject<CLHeading, Error> ②
    var publisher: AnyPublisher<CLHeading, Error> ③

    override init() {
        mgr = CLLocationManager()
        headingPublisher = PassthroughSubject<CLHeading, Error>()
        publisher = headingPublisher.eraseToAnyPublisher()

        super.init()
        mgr.delegate = self ④
    }

    func enable() {
        mgr.startUpdatingHeading() ⑤
    }

    func disable() {
        mgr.stopUpdatingHeading()
    }
    // MARK - delegate methods

    /*
     *  locationManager:didUpdateHeading:
     *
     *  Discussion:
     *    Invoked when a new heading is available.
     */
    func locationManager(_ manager: CLLocationManager, didUpdateHeading newHeading:
CLHeading) {
        headingPublisher.send(newHeading) ⑥
    }

    /*
     *  locationManager:didFailWithError:
     *  Discussion:
     *    Invoked when an error has occurred. Error types are defined in "CLError.h".
     */
    func locationManager(_ manager: CLLocationManager, didFailWithError error: Error)
{
        headingPublisher.send(completion: Subscribers.Completion.failure(error)) ⑦
    }
}
```

① CLLocationManager is the heart of what is being wrapped, part of CoreLocation. Because it has additional methods that need to be called for using the framework, I exposed it as a read-only, but public, property. An example of this need is for requesting user permission to use the location API, which the framework exposes as a method on CLLocationManager.

② A private instance of PassthroughSubject with the data type we want to publish provides our inside-the-class access to forward data.

③ An the public property `publisher` exposes the publisher from that subject for external subscriptions.

④ The heart of this works by assigning this class as the delegate to the CLLocationManager instance, which is set up at the tail end of initialization.

⑤ The CoreLocation API doesn't immediately start sending information. There are methods that need to be called to start (and stop) the data flow, and these are wrapped and exposed on this proxy object. Most publishers are set up to subscribe and drive consumption based on subscription, so this is a bit out of the norm for how a publisher starts generating data.

⑥ With the delegate defined and the CLLocationManager activated, the data will be provided via callbacks defined on the CLLocationManagerDelegate. We implement the callbacks we want for this wrapped object, and within them we use passthroughSubject `.send()` to forward the information to any existing subscribers.

⑦ While not strictly required, the delegate provided an Error reporting callback, so we included that as an example of forwarding an error through passthroughSubject.

UIKit-Combine/HeadingViewController.swift

```
import UIKit
import Combine
import CoreLocation

class HeadingViewController: UIViewController {

    var headingSubscriber: AnyCancellable?

    let coreLocationProxy = LocationHeadingProxy()
    var headingBackgroundQueue: DispatchQueue = DispatchQueue(label:
"headingBackgroundQueue")

    // MARK - lifecycle methods

    @IBOutlet weak var permissionButton: UIButton!
    @IBOutlet weak var activateTrackingSwitch: UISwitch!
    @IBOutlet weak var headingLabel: UILabel!
    @IBOutlet weak var locationPermissionLabel: UILabel!

    @IBAction func requestPermission(_ sender: UIButton) {
        print("requesting corelocation permission")
        let _ = Future<Int, Never> { promise in ①
            self.coreLocationProxy.mgr.requestWhenInUseAuthorization()
            return promise(.success(1))
```

```
        }
        .delay(for: 2.0, scheduler: headingBackgroundQueue) ②
        .receive(on: RunLoop.main)
        .sink { _ in
            print("updating corelocation permission label")
            self.updatePermissionStatus() ③
        }
    }

    @IBAction func trackingToggled(_ sender: UISwitch) {
        switch sender.isOn {
        case true:
            self.coreLocationProxy.enable() ④
            print("Enabling heading tracking")
        case false:
            self.coreLocationProxy.disable()
            print("Disabling heading tracking")
        }
    }

    func updatePermissionStatus() {
        let x = CLLocationManager.authorizationStatus()
        switch x {
        case .authorizedWhenInUse:
            locationPermissionLabel.text = "Allowed when in use"
        case .notDetermined:
            locationPermissionLabel.text = "notDetermined"
        case .restricted:
            locationPermissionLabel.text = "restricted"
        case .denied:
            locationPermissionLabel.text = "denied"
        case .authorizedAlways:
            locationPermissionLabel.text = "authorizedAlways"
        @unknown default:
            locationPermissionLabel.text = "unknown default"
        }
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.

        // request authorization for the corelocation data
        self.updatePermissionStatus()

        let corelocationsub = coreLocationProxy
            .publisher
            .print("headingSubscriber")
            .receive(on: RunLoop.main)
            .sink { someValue in ⑤
                self.headingLabel.text = String(someValue.trueHeading)
```

```
            }
        headingSubscriber = AnyCancellable(corelocationsub)
    }

}
```

① One of the quirks of CoreLocation is the requirement to ask for permission from the user to access the data. The API provided to initiate this request returns immediately, but provides no detail if the user allowed or denied the request. The CLLocationManager class includes the information, and exposes it as a class method when you want to retrieve it, but there is no information provided to know when, or if, the user has responded to the request. Since the operation doesn't provide any return, we provide an integer as the pipeline data, primarily to represent that the request has been made.

② Since there isn't a clear way to judge, but the permission is persistent, we simply use a delay operator before attempting to retrieve the data. This use simply delays the propogation of the value for two seconds.

③ After that delay, we invoke the class method and attempt to update informtion in the interface with the results of the current provided status.

④ Since CoreLocation requires methods to be explicitly enabled or disabled to provide the data, this connects a UISwitch toggle IBAction to the methods exposed on our publisher proxy.

⑤ The heading data is received in this sink subscriber, where in this example we simply write it to a text label.

Responding to updates from NotificationCenter

Goal

- Receiving notifications from NotificationCenter as a publisher to declaratively react to the information provided.

References

- NotificationCenter

See also

- The unit tests at `UsingCombineTests/NotificationCenterPublisherTests.swift`

Code and explanation

A large number of frameworks and user interface components provide information about their state and interactions via Notifications from NotificationCenter. Apple's documentation includes an article on receiving and handling events with Combine specifically referencing NotificationCenter.

Notifications flowing through NotificationCenter provide a common, central location for events within your application.

You can also add your own notifications to your application, and upon sending them may include an additional dictionary in their `userInfo` property. An example of defining your own notification `.myExampleNotification`:

```
extension Notification.Name {
    static let myExampleNotification = Notification.Name("an-example-notification")
}
```

Notification names are structured, and based on Strings. Object references can be passed when a notification is posted to the NotificationCenter, indicating which object sent the notification. Additionally, Notifications may include a `userInfo`, which has a type of `[AnyHashable : Any]?`. This allows for arbitrary dictionaries, either reference or value typed, to be included with a notification.

```
let myUserInfo = ["foo": "bar"]

let note = Notification(name: .myExampleNotification, userInfo: myUserInfo)
NotificationCenter.default.post(note)
```

While commonly in use within AppKit and MacOS applications, not all developers are comfortable with heavily using NotificationCenter. Originating within the more dynamic Objective-C runtime, Notifications leverage Any and optional types quite extensively. Using them within Swift code, or a pipeline, implies that the pipeline will have to provide the type assertions and deal with any possible errors related to data that may or may not be expected.

When creating the NotificationCenter publisher, you provide the name of the notification for which

you want to receive, and optionally an object reference to filter to specific types of objects. A number of AppKit components that are subclasses of NSControl share a set of notifications, and filtering can be critical to getting the right notification.

An example of subscribing to AppKit generated notifications:

```
let sub = NotificationCenter.default.publisher(for: NSControl
.textDidChangeNotification, ①
                                                object: filterField) ②
    .map { ($0.object as! NSTextField).stringValue }  ③
    .assign(to: \MyViewModel.filterString, on: myViewModel) ④
```

① TextFields within AppKit generate textDidChangeNotifications when the values are updated.

② An AppKit application can frequently have a large number of text fields that may be changed. Including a reference to the sending control can be used to filter to text changed notifications to which you are specifically interested in responding.

③ the map operator can be used to get into the object references included with the notification, in this case the `.stringValue` property of the text field that sent the notification, providing its updated value

④ The resulting string can be assigned using a writable KeyValue path.

An example of subscribing to your own notifications:

```
let cancellable = NotificationCenter.default.publisher(for: .myExampleNotification,
object: nil)
    // can't use the object parameter to filter on a value reference, only class
references, but
    // filtering on 'nil' only constrains to notification name, so value objects *can*
be passed
    // in the notification itself.
    .sink { receivedNotification in
        print("passed through: ", receivedNotification)
        // receivedNotification.name
        // receivedNotification.object - object sending the notification (sometimes
nil)
        // receivedNotification.userInfo - often nil
    }
```

SwiftUI Integration

Using BindableObject with SwiftUI models as a publisher source

Goal

- SwiftUI includes @Binding and the BindableObject protocol, which provides a publishing source to alerts to model objects changing.

References

- << link to reference pages>>

See also

- << link to other patterns>>

Code and explanation

Testing and Debugging

The Publisher/Subscriber interface in Combine is beautifully suited to be an easily testable interface.

With the composability of Combine, you can use this to your advantage, creating APIs that present, or consume, code that conforms to Publisher.

With the publisher protocol as the key interface, you can replace either side to validate your code in isolation.

For example, if your code was focused on providing its data from external web services through Combine, you might make the interface to this conform to `AnyPublisher<Data, Error>`. You could then use that interface to test either side of that pipeline independently.

- You could mock data responses that emulate the underlying API calls and possible responses, including various error conditions. This might include returning data from a publisher created with Just or Fail, or something more complex using Future. None of these options require you to make actual network interface calls.
- Likewise you can isolate the testing of making the publisher do the API calls and verify the various success and failure conditions expected.

Testing a publisher with XCTestExpectation

Goal

- For testing a publisher (and any pipeline attached)

References

- UsingCombineTests/DataTaskPublisherTests.swift
- UsingCombineTests/EmptyPublisherTests.swift
- UsingCombineTests/FuturePublisherTests.swift
- UsingCombineTests/PublisherTests.swift
- UsingCombineTests/DebounceAndRemoveDuplicatesPublisherTests.swift

See also

- Testing a publisher with XCTestExpectation
- Testing a subscriber with scheduled sends from PassthroughSubject
- Testing a subscriber with a PassthroughSubject

Code and explanation

When you are testing a publisher, or something that creates a publisher, you may not have the option of controlling when the publisher returns data for your tests. Combine, being driven by its subscribers, can set up a sync that initiates the data flow. You can use an XCTestExpectation to wait an explicit amount of time for the test to run to completion.

A general pattern for using this with Combine includes:

1. set up the expectation within the test
2. establish the code you are going to test
3. set up the code to be invoked such that on the success path you call the expectation's `.fulfill()` function
4. set up a `wait()` function with an explicit timeout that will fail the test if the expectation isn't fulfilled within that time window.

If you are testing the data results from a pipeline, then triggering the `fulfill()` function within the sink operator `receiveValue` closure can be very convenient. If you are testing a failure condition from the pipeline, then often including `fulfill()` within the sink operator `receiveCompletion` closure is effective.

The following example shows testing a one-shot publisher (URLSession.dataTaskPublisher in this case) using expectation, and expecting the data to flow without an error.

UsingCombineTests/DataTaskPublisherTests.swift - testDataTaskPublisher

```
func testDataTaskPublisher() {
        // setup
        let expectation = XCTestExpectation(description: "Download from \(String
(describing: testURL))") ①
        let remoteDataPublisher = URLSession.shared.dataTaskPublisher(for: self
.testURL!)
            // validate
            .sink(receiveCompletion: { fini in
                print(".sink() received the completion", String(describing: fini))
                switch fini {
                case .finished: expectation.fulfill() ②
                case .failure: XCTFail() ③
                }
            }, receiveValue: { (data, response) in
                guard let httpResponse = response as? HTTPURLResponse else {
                    XCTFail("Unable to parse response an HTTPURLResponse")
                    return
                }
                XCTAssertNotNil(data)
                // print(".sink() data received \(data)")
                XCTAssertNotNil(httpResponse)
                XCTAssertEqual(httpResponse.statusCode, 200) ④
                // print(".sink() httpResponse received \(httpResponse)")
            })

        XCTAssertNotNil(remoteDataPublisher)
        wait(for: [expectation], timeout: 5.0) ⑤
    }
```

① The expectation is set up with a string that makes debugging in the event of failure a bit easier. This string is really only seen when a test failure occurs. The code we are testing here is dataTaskPublisher retrieving data from a preset test URL, defined earlier in the test. The

publisher is invoked by attaching the sink subscriber to it. Without the expectation, the code will still run, but the test running structure wouldn't wait to see if there were any exceptions. The expectation within the test "holds the test" waiting for a response to let the operators do their work.

② In this case, the test is expected to complete successfully and terminate normally, therefore the `expectation.fulfill()` invocation is set within the receiveCompletion closure, specifically linked to a received `.finished` completion.

③ Since we don't expect a failure, we also have an explicit XCTFail() invocation if we receive a `.failure` completion.

④ We have a few additional assertions within the receiveValue. Since this publisher set returns a single value and then terminates, we can make easily make inline assertions about the data received. If we received multiple values, then we could collect those and make assertions on what was received after the fact.

⑤ This test uses a single expectation, but you can include multiple independent expectations to require fulfillment. It also sets that maximum time that this test can run to five seconds. The test will not always take five seconds, as it will complete the test as soon as the fulfill is received.

Testing a subscriber with a PassthroughSubject

Goal

- For testing a subscriber, or something that includes a subscriber, we can emulate the publishing source with PassthroughSubject to provide explicit control of what data gets sent and when.

References

- UsingCombineTests/EncodeDecodeTests.swift
- UsingCombineTests/FilterPublisherTests.swift
- UsingCombineTests/FuturePublisherTests.swift
- UsingCombineTests/RetryPublisherTests.swift
- UsingCombineTests/SinkSubscriberTests.swift
- UsingCombineTests/SwitchAndFlatMapPublisherTests.swift
- UsingCombineTests/DebounceAndRemoveDuplicatesPublisherTests.swift

See also

- Testing a publisher with XCTestExpectation
- passthroughSubject
- Testing a subscriber with scheduled sends from PassthroughSubject
- Using EntwineTest to create a testable publisher and subscriber

Code and explanation

When you are testing a subscriber in isolation, you can often get more fine-grained control of your tests by emulating the publisher with a passthroughSubject and using the associated `.send()` method to trigger controlled updates.

This pattern relies on the subscriber setting up the initial part of the publisher-subscriber lifecycle upon construction, and leaving the code to stand waiting until data is provided. With a PassthroughSubject, sending the data to trigger the pipeline and subscriber closures, or following state changes that can be verified, is at the control of the test code itself.

This kind of testing pattern also works well when you are testing the response of the subscriber to a failure, which might otherwise terminate a subscription.

A general pattern for using this kind of test construct is:

1. set up your subscriber and any pipeline leading to it that you want to include within the test
2. create a PassthroughSubject in the test that produces an output type and failure type to match with your subscriber.
3. assert any initial values or preconditions
4. send the data through the subject
5. test the results of having sent the data - either directly or asserting on state changes that were expected

6. send additional data if desired
7. test further evolution of state or other changes.

An example of this pattern follows:

UsingCombineTests/SinkSubscriberTests.swift - testSinkReceiveDataThenError

```
func testSinkReceiveDataThenError() {

    // setup - preconditions ①
    let expectedValues = ["firstStringValue", "secondStringValue"]
    enum TestFailureCondition: Error {
        case anErrorExample
    }
    var countValuesReceived = 0
    var countCompletionsReceived = 0

    // setup
    let simplePublisher = PassthroughSubject<String, Error>() ②

    let _ = simplePublisher ③
        .sink(receiveCompletion: { completion in
            countCompletionsReceived += 1
            switch completion { ④
            case .finished:
                print(".sink() received the completion:", String(describing:
completion))
                // no associated data, but you can react to knowing the
                // request has been completed
                XCTFail("We should never receive the completion, the error should
happen first")
                break
            case .failure(let anError):
                // do what you want with the error details, presenting,
                // logging, or hiding as appropriate
                print("received the error: ", anError)
                XCTAssertEqual(anError.localizedDescription,
                               TestFailureCondition.anErrorExample
.localizedDescription) ⑤
                break
            }
        }, receiveValue: { someValue in ⑥
            // do what you want with the resulting value passed down
            // be aware that depending on the data type being returned,
            // you may get this closure invoked multiple times.
            XCTAssertNotNil(someValue)
            XCTAssertTrue(expectedValues.contains(someValue))
            countValuesReceived += 1
            print(".sink() received \(someValue)")
        })
```

```
    // validate
    XCTAssertEqual(countValuesReceived, 0) ⑦
    XCTAssertEqual(countCompletionsReceived, 0)

    simplePublisher.send("firstStringValue") ⑧
    XCTAssertEqual(countValuesReceived, 1)
    XCTAssertEqual(countCompletionsReceived, 0)

    simplePublisher.send("secondStringValue")
    XCTAssertEqual(countValuesReceived, 2)
    XCTAssertEqual(countCompletionsReceived, 0)

    simplePublisher.send(completion: Subscribers.Completion.failure
(TestFailureCondition.anErrorExample))  ⑨
    XCTAssertEqual(countValuesReceived, 2)
    XCTAssertEqual(countCompletionsReceived, 1)

    // this data will never be seen by anything in the pipeline above because we've
already sent a completion
    simplePublisher.send(completion: Subscribers.Completion.finished) ⑩
    XCTAssertEqual(countValuesReceived, 2)
    XCTAssertEqual(countCompletionsReceived, 1)
}
```

① This test sets up some variables to capture and modify during test execution that we use to validate when and how the sink code operates. Additionally, we have an error definition defined here because it's not coming from other code elsewhere.

② The setup for this code uses the passthroughSubject to drive the test, but the code we're interested in testing is really the subscriber.

③ The subscriber setup under test (in this case, a standard sink). We have code paths that trigger on receiving data and completions.

④ Within the completion path, we switch on the type of completion, adding an assertion that will fail the test if a finish is called, as we expect to only generate a `.failure` completion.

⑤ I find testing error equality in Swift to be awkward, but if the error is code you are controller, you can sometimes use the `localizedDescription` as a convenient way to test the type of error received.

⑥ The `receiveValue` closure is more complex in how it asserts its values. Since we are receiving multiple values in the process of this test, we have some additional logic to simply check that the values are within the set that we send. Like the completion handler, We also increment test specific variables that we will assert on later to validate state and order of operation.

⑦ The count variables are validated as preconditions before we send any data to double check our assumptions.

⑧ In the test, the `send()` triggers the actions, and immediately after we can test the side effects through the test variables we are updating. In your own code, you may not be able to (or want to) modify your subscriber, but you may be able to provide private/testable properties or windows into the objects to validate them in a similiar fashion.

⑨ We also use `send()` to trigger a completion, in this case a failure completion.

⑩ And the final `send()` is simply validating the operation of the failure that just happened - that it wasn't processed, and no further state updates happened.

Testing a subscriber with scheduled sends from PassthroughSubject

Goal

- For testing a pipeline, or subscriber, when part of what you want to test is the timing of the pipeline.

References

- UsingCombineTests/PublisherTests.swift
- UsingCombineTests/FuturePublisherTests.swift
- UsingCombineTests/SinkSubscriberTests.swift
- UsingCombineTests/SwitchAndFlatMapPublisherTests.swift
- UsingCombineTests/DebounceAndRemoveDuplicatesPublisherTests.swift

See also

- Testing a subscriber with a PassthroughSubject
- Using EntwineTest to create a testable publisher and subscriber
- Testing a publisher with XCTestExpectation
- passthroughSubject

Code and explanation

> There are a number of operators in Combine that are specific to the timing of data, including debounce, throttle, and delay. You may want to test that your pipeline timing is having the desired impact, indepedently of doing UI testing.

One way of handling this leverages the both XCTestExpectation and a passthroughSubject, combining the two. Building on both Testing a publisher with XCTestExpectation and Testing a subscriber with a PassthroughSubject, add DispatchQueue in the test to schedule invocations of PassthroughSubject's `.send()` method.

An example of this:

UsingCombineTests/PublisherTests.swift - testKVOPublisher

```
func testKVOPublisher() {
    let expectation = XCTestExpectation(description: self.debugDescription)
    let foo = KVOAbleNSObject()
    let q = DispatchQueue(label: self.debugDescription) ①

    let _ = foo.publisher(for: \.intValue)
        .print()
        .sink { someValue in
            print("value of intValue updated to: >>\(someValue)<<")
        }

    q.asyncAfter(deadline: .now() + 0.5, execute: { ②
        print("Updating to foo.intValue on background queue")
        foo.intValue = 5
        expectation.fulfill() ③
    })
    wait(for: [expectation], timeout: 5.0) ④
}
```

① This adds a DispatchQueue to your test, conveniently naming the queue after the test itself. This really only shows when debugging test failures, and is convenient as a reminder of what's happening in the test code vs. any other background queues that might be in use.

② `.asyncAfter` is used along with the deadline parameter to define when a call gets made.

③ The simplest form embeds any relevant assertions into the subscriber or around the subscriber. Additionally, invoking the `.fulfill()` on your expectation as the last queued entry you send lets the test know that it is now complete.

④ Make sure that when you set up the wait that allow for sufficient time for your queue'd calls to be invoked.

A definite downside to this technique is that it forces the test to take a minimum amount of time matching the maximum queue delay in the test.

Another option is a 3rd party library named EntwineTest, which was inspired by the RxTest library. EntwineTest is part of Entwine, a Swift library that expands on Combine with some helpers. The library can be found on Github at https://github.com/tcldr/Entwine.git, available under the MIT license.

One of the key elements included in EnwtineTest is a virtual time scheduler, as well as additional classes that schedule (TestablePublisher) and collect and record (TestableSubscriber) the timing of results while using this scheduler.

An example of this from the EntwineTest project README is included:

UsingCombineTests/EntwineTestExampleTests.swift - testExampleUsingVirtualTimeScheduler

```
func testExampleUsingVirtualTimeScheduler() {
    let scheduler = TestScheduler(initialClock: 0) ①
    var didSink = false
    let cancellable = Just(1) ②
        .delay(for: 1, scheduler: scheduler)
        .sink { _ in
            didSink = true
        }

    XCTAssertNotNil(cancellable)
    // where a real scheduler would have triggered when .sink() was invoked
    // the virtual time scheduler requires resume() to commence and runs to
    // completion.
    scheduler.resume() ③
    XCTAssertTrue(didSink) ④
}
```

① Using the virtual time scheduler requires you create one at the start of the test, initializing its clock to a starting value. The virtual time scheduler in EntwineTest will commence subscription at the value `200` and times out at `900` if the pipeline isn't complete by that time.

② You create your pipeline, along with any publishers or subscribers, as normal. EntwineTest also offers a testable publisher and a testable subscriber that could be used as well. For more details on these parts of EntwineTest, see Using EntwineTest to create a testable publisher and subscriber.

③ `.resume()` needs to be invoked on the virtual time scheduler to commence its operation and run the pipeline.

④ Assert against expected end results after the pipeline has run to completion.

Using EntwineTest to create a testable publisher and subscriber

Goal

- For testing a pipeline, or subscriber, when part of what you want to test is the timing of the pipeline.

References

- UsingCombineTests/EntwineTestExampleTests.swift

See also

- Testing a publisher with XCTestExpectation
- Testing a subscriber with a PassthroughSubject
- Testing a subscriber with scheduled sends from PassthroughSubject
- passthroughSubject

Code and explanation

The EntwineTest library, available from Gitub at https://github.com/tcldr/Entwine.git, provides some additional options for making your pipelines testable. In addition to a virtual time scheduler, EntwineTest has a TestablePublisher and a TestableSubscriber. These work in coordination with the virtual time scheduler to allow you to specify the timing of the publisher generating data, and to valid the data received by the subscriber.

As of Xcode 11.2, there is a bug with SwiftPM that impacts the use of Entwine as a testing library. The details can be found at SR-11564 in Swift's open source bug reporting.

The workaround, which you may need to apply if using Xcode 11.2, is to set the project setting `DEAD_CODE_STRIPPING=NO`.

An example of this from the EntwineTest project is included:

UsingCombineTests/EntwineTestExampleTests.swift - testMap

```
import XCTest
import EntwineTest
// library loaded from
https://github.com/tcldr/Entwine/blob/master/Assets/EntwineTest/README.md
// as a Swift package https://github.com/tcldr/Entwine.git : 0.6.0, Next Major Version

class EntwineTestExampleTests: XCTestCase {

    func testMap() {

        let testScheduler = TestScheduler(initialClock: 0)

        // creates a publisher that will schedule its elements relatively, at the
point of subscription
        let testablePublisher: TestablePublisher<String, Never> = testScheduler
.createRelativeTestablePublisher([ ①
            (100, .input("a")),
            (200, .input("b")),
            (300, .input("c")),
        ])

        // a publisher that maps strings to uppercase
        let subjectUnderTest = testablePublisher.map { $0.uppercased() }

        // uses the method described above (schedules a subscription at 200, to be
cancelled at 900)
        let results = testScheduler.start { subjectUnderTest } ②

        XCTAssertEqual(results.recordedOutput, [ ③
            (200, .subscription),           // subscribed at 200
            (300, .input("A")),             // received uppercased input @ 100 +
subscription time
            (400, .input("B")),             // received uppercased input @ 200 +
subscription time
            (500, .input("C")),             // received uppercased input @ 300 +
subscription time
        ])
    }
}
```

① The TestablePublisher lets you set up a publisher that returns specific values at specific times. In this case, it's returning 3 items at consistent intervals.

② When you use the virtual time scheduler, it is important to make sure to invoke it with start. This runs the virtual time scheduler, which can run faster than a clock since it only needs to increment the virtual time and not wait for elapsed time.

③ `results` is a TestableSubscriber object, and includes a `recordedOutput` property which provides an ordered list of all the data and combine control path interactions with their timing.

If this test sequence had been done with asyncAfter, then the test would have taken a minimum of 500ms to complete. When I ran this test on my laptop, it was recording 0.0121 seconds to complete the test (12.1ms).

Debugging pipelines with the print operator

Goal

- To gain understanding of what is happening in a pipeline, seeing all control and data interactions.

References

- print
- sink
- retry
- The ViewController with this code is in the github project at UIKit-Combine/GithubViewController.swift
- The retry unit tests in the github project at UsingCombineTests/RetryPublisherTests.swift

See also

- Cascading UI updates including a network request
- Sequencing operations with Combine
- Declarative UI updates from user input
- Debugging pipelines with the debugger
- Debugging pipelines with the handleEvents operator

Code and explanation

I have found the greatest detail of information comes from selectively using the print operator. The downside is that it prints quite a lot of information, so the output can become quickly overwhelming. For understanding a simple pipeline, using the `.print()` as an operator without any parameters is very straightforward. As soon as you want to add more than one print operator, you will likely want to use the string parameter, which is puts in as a prefix to the output.

The example Cascading UI updates including a network request uses it in several places, with long descriptive prefixes to make it clear which pipeline is providing the information.

The two pipelines cascade together by connecting through a private published variable - the github user data. The two relevant pipelines from that example code:

UIKit-Combine/GithubViewController.swift

```
usernameSubscriber = $username
    .throttle(for: 0.5, scheduler: myBackgroundQueue, latest: true)
    // ^^ scheduler myBackGroundQueue publishes resulting elements
    // into that queue, resulting on this processing moving off the
    // main runloop.
    .removeDuplicates()
    .print("username pipeline: ") // debugging output for pipeline
    .map { username -> AnyPublisher<[GithubAPIUser], Never> in
        return GithubAPI.retrieveGithubUser(username: username)
    }
    // ^^ type returned in the pipeline is a Publisher, so we use
    // switchToLatest to flatten the values out of that
    // pipeline to return down the chain, rather than returning a
    // publisher down the pipeline.
    .switchToLatest()
    // using a sink to get the results from the API search lets us
    // get not only the user, but also any errors attempting to get it.
    .receive(on: RunLoop.main)
    .assign(to: \.githubUserData, on: self)

// using .assign() on the other hand (which returns an
// AnyCancellable) *DOES* require a Failure type of <Never>
repositoryCountSubscriber = $githubUserData
    .print("github user data: ")
    .map { userData -> String in
        if let firstUser = userData.first {
            return String(firstUser.public_repos)
        }
        return "unknown"
    }
    .receive(on: RunLoop.main)
    .assign(to: \.text, on: repositoryCountLabel)
```

When you run the UIKit-Combine example code, the terminal shows the following output as I slowly enter the username `heckj`. In the course of doing these lookups, two other github accounts are found and retrieved (`hec` and `heck`) before the final one.

interactive output from simulator

```
username pipeline: : receive subscription: (RemoveDuplicates)
username pipeline: : request unlimited
github user data: : receive subscription: (CurrentValueSubject)
github user data: : request unlimited
github user data: : receive value: ([])
username pipeline: : receive value: ()
github user data: : receive value: ([])

Set username to  h
username pipeline: : receive value: (h)
github user data: : receive value: ([])

Set username to  he
username pipeline: : receive value: (he)
github user data: : receive value: ([])

Set username to  hec
username pipeline: : receive value: (hec)

Set username to  heck
github user data: : receive value: ([UIKit_Combine.GithubAPIUser(login: "hec",
public_repos: 3, avatar_url: "https://avatars3.githubusercontent.com/u/53656?v=4")])

username pipeline: : receive value: (heck)
github user data: : receive value: ([UIKit_Combine.GithubAPIUser(login: "heck",
public_repos: 6, avatar_url: "https://avatars3.githubusercontent.com/u/138508?v=4")])

Set username to  heckj
username pipeline: : receive value: (heckj)
github user data: : receive value: ([UIKit_Combine.GithubAPIUser(login: "heckj",
public_repos: 69, avatar_url: "https://avatars0.githubusercontent.com/u/43388?v=4")])
```

I have removed some of the other extraneous print statements, which I also placed in sink closures to see final results.

As you can see, you see the initial subscription setup at the very beginning, and then notifications, including the debug representation of the value passed through the print operator. Although it is not shown in the example content above, you will also see cancellations when an error occurs, or completions when they emit from a publisher reporting no further data is available.

It can also be beneficial to use a print operator on either side of an operator to understand how it is operating.

An example of doing this, leveraging the prefix to show the retry operator and how it works:

UsingCombineTests/RetryPublisherTests.swift

```
func testRetryWithOneShotFailPublisher() {
    // setup

    let _ = Fail(outputType: String.self, failure: TestFailureCondition
.invalidServerResponse)
        .print("(1)>") ①
        .retry(3)
        .print("(2)>") ②
        .sink(receiveCompletion: { fini in
            print(" ** .sink() received the completion:", String(describing: fini))
        }, receiveValue: { stringValue in
            XCTAssertNotNil(stringValue)
            print(" ** .sink() received \(stringValue)")
        })
}
```

① The `(1)` prefix is to show the interactions above the retry operator

② The `(2)` prefix shows the interactions after the retry operator

output from unit test

```
Test Suite 'Selected tests' started at 2019-07-26 15:59:48.042
Test Suite 'UsingCombineTests.xctest' started at 2019-07-26 15:59:48.043
Test Suite 'RetryPublisherTests' started at 2019-07-26 15:59:48.043
Test Case '-[UsingCombineTests.RetryPublisherTests testRetryWithOneShotFailPublisher]'
started.
(1)>: receive subscription: (Empty) ①
(1)>: receive error: (invalidServerResponse)
(1)>: receive subscription: (Empty)
(1)>: receive error: (invalidServerResponse)
(1)>: receive subscription: (Empty)
(1)>: receive error: (invalidServerResponse)
(1)>: receive subscription: (Empty)
(1)>: receive error: (invalidServerResponse)
(2)>: receive error: (invalidServerResponse) ②
 ** .sink() received the completion:
failure(UsingCombineTests.RetryPublisherTests.TestFailureCondition.invalidServerRespon
se)
(2)>: receive subscription: (Retry)
(2)>: request unlimited
(2)>: receive cancel
Test Case '-[UsingCombineTests.RetryPublisherTests testRetryWithOneShotFailPublisher]'
passed (0.010 seconds).
Test Suite 'RetryPublisherTests' passed at 2019-07-26 15:59:48.054.
     Executed 1 test, with 0 failures (0 unexpected) in 0.010 (0.011) seconds
Test Suite 'UsingCombineTests.xctest' passed at 2019-07-26 15:59:48.054.
     Executed 1 test, with 0 failures (0 unexpected) in 0.010 (0.011) seconds
Test Suite 'Selected tests' passed at 2019-07-26 15:59:48.057.
     Executed 1 test, with 0 failures (0 unexpected) in 0.010 (0.015) seconds
```

① In the test sample, the publisher always reports a failure, resulting in seeing the prefix `(1)` receiving the error, and then the resubscription from the retry operator.

② And after 4 of those attempts (3 "retries"), then you see the error falling through the pipeline. After the error hits the sink, you see the `cancel` signal propogated back up, which stops at the retry operator.

While very effective, the print operator can be a blunt tool, generating a lot of output that you have to parse and review. If you want to be more selective with what you identify and print, or if you need to process the data passing through for it to be used more meaningfully, then you look at the handleEvents operator. More detail on how to use this opeartor for debugging is in Debugging pipelines with the handleEvents operator.

Debugging pipelines with the handleEvents operator

Goal

- To get more targettted understanding of what is happening within a pipeline, employing breakpoints, print or logging statements, or additional logic.

References

- handleEvents
- A ViewController using handleEvents is in the github project at UIKit-Combine/GithubViewController.swift
- The handleEvents unit tests in the github project at UsingCombineTests/HandleEventsPublisherTests.swift

See also

- Debugging pipelines with the print operator
- Cascading UI updates including a network request
- Sequencing operations with Combine
- Declarative UI updates from user input
- Debugging pipelines with the debugger

Code and explanation

> handleEvents passes data through, making no modifications to the output and failure types, or the data. When you put in the operator, you can specify a number of optional closures, allowing you to focus on the aspect of what you want to see. The handleEvents operator with specific closures can be a great way to get a window to see what is happening when a pipeline is cancelling, erroring, or otherwise terminating expectedly.

The closures you can provide include:

- `receiveSubscription`
- `receiveRequest`
- `receiveCancel`
- `receiveOutput`
- `receiveCompletion`

If the closures each included a print statement, then this operator would be acting very much like the print operator, as detailed in Debugging pipelines with the print operator.

The power of handleEvents for debugging is in selecting what you want to view, reducing the amount of output, or manipulating the data to get a better understanding of it.

In the example viewcontroller at UIKit-Combine/GithubViewController.swift, the subscription, cancellation, and completion handlers are used to provide a side effect of starting, or stopping, an activity indicator.

If you only wanted to see the data being passed on the pipeline, and didn't care about the control messages, then providing a single closure for receiveOutput and ignoring the other closures can let you focus in on just that detail.

The unit test example showing handleEvents has each active with comments:

UsingCombineTests/HandleEventsPublisherTests.swift

```
.handleEvents(receiveSubscription: { aValue in
    print("receiveSubscription event called with \(String(describing: aValue))") ②
}, receiveOutput: { aValue in ③
    print("receiveOutput was invoked with \(String(describing: aValue))")
}, receiveCompletion: { aValue in ④
    print("receiveCompletion event called with \(String(describing: aValue))")
}, receiveCancel: { ⑤
    print("receiveCancel event invoked")
}, receiveRequest: { aValue in ①
    print("receiveRequest event called with \(String(describing: aValue))")
})
```

① The first closure called is `receiveRequest`, which will have the demand value passed into it.

② The second closure `receiveSubscription` is commonly the returning subscription from the publisher, which passes in a reference to the publisher. At this point, the pipeline is operational, and the publisher will provide data based on the amount of data requested in the original request.

③ This data is passed into `receiveOutput` as the publisher makes it available, invoking the closure for each value passed. This will repeat for as many values as the publisher sends.

④ If the pipeline is closed - either normally or terminated due to a failure - the `receiveCompletion` closure will get the completion. Just the like the sink closure, you can switch on the completion provided, and if it is a `.failure` completion, then you can inspect the enclosed error.

⑤ If the pipeline is cancelled, then the `receiveCancel` closure will be called. No data is passed into the cancellation closure.

While you can also use breakpoint and breakpointOnError operators to break into a debugger (as shown in Debugging pipelines with the debugger), the handleEvents() operator with closures allows you to set breakpoints within Xcode. This allows you to immediately jump into the debugger to inspect the data flowing through the pipeline, or to get references to the subscriber, or the error in the case of a failed completion.

Debugging pipelines with the debugger

Goal

- To force the pipeline to trap into a debugger on specific scenarios or conditions.

References

- handleEvents
- map

See also

- Debugging pipelines with the print operator
- Debugging pipelines with the handleEvents operator

Code and explanation

You can easily set a breakpoint within any closure to any operator within a pipeline, triggering the debugger to activate to inspect the data. Since the map operator is frequently used for simple output type conversions, it is often an excellent candidate that has a closure you can use. If you want to see into the control messages, then a breakpoint within any of the closures provided to handleEvents makes a very convenient target.

You can also use the breakpoint operator to trigger the debugger, which can be a very quick and convenient way to see what is happening in a pipeline. The breakpoint operator acts very much like handleEvents, taking a number of optional parameters, closures that are expected to return a boolean, and if true will invoke the debugger.

The optional closures include:

- `receiveSubscription`
- `receiveOutput`
- `receiveCompletion`

```
.breakpoint(receiveSubscription: { subscription in
    return false // return true to throw SIGTRAP and invoke the debugger
}, receiveOutput: { value in
    return false // return true to throw SIGTRAP and invoke the debugger
}, receiveCompletion: { completion in
    return false // return true to throw SIGTRAP and invoke the debugger
})
```

This allows you to provide logic to evaluate the data being passed through, and only triggering a breakpoint when your specific conditions are met. With very active pipelines processing a lot of data, this can be a great tool to be more surgical in getting the debugger active when you need it, and letting the other data move on by.

If you are only interested in the breaking into the debugger on error conditions, then convenience operator breakPointOnError is perfect. It takes no parameters or closures, simply invoking the debugger when an error condition of any form is passed through the pipeline.

```
.breakpointOnError()
```

The location of the breakpoint that is triggered by the breakpoint operator isn't in your code, so getting to local frames and information can be a bit tricky. This does allow you to inspect global application state in highly specific instances (whenever the closure returns `true`, with logic you provide), but you may find it more effective to use regular breakpoints within closures. The breakpoint() and breakpointOnError() operators don't immediately drop you into a closure where you can see the data being passed, error thrown, or control signals that may have triggered the breakpoint. You can often walk back up the stack trace within the debugging window to see the publisher.

When you trigger a breakpoint within an operator's closure, the debugger immediately gets the context of that closure as well, so you can see/inspect the data being passed.

Reference

reference preamble goes here…

This is intended to extend Apple's documentation, not replace it.

- The documentation associated with iOS 13 GM.

things to potentially include for each segment

- narrative description of what the function does
  - notes on why you might want to use it, or where you may see it
  - xref back to patterns document where functions are being used
- marble/railroad diagram explaining what the transformation/operator does
- sample code showing it being used and/or tested

Publishers

For general information about publishers see Publishers and Lifecycle of Publishers and Subscribers.

Just

Summary

`Just` provides a single result and then terminates, providing a publisher with a failure type of `<Never>`

docs

`Just`

Usage

- Using catch to handle errors in a one-shot pipeline
- Using flatMap with catch to handle errors
- Declarative UI updates from user input
- Cascading UI updates including a network request

Details

Often used within a closure to flatMap in error handling, it can see a one-shot pipeline for use in error handling of continuous values.

Future

Summary

A future is initialized with a closure that eventually resolves to a single output value or failure completion.

docs

Future.

Usage

- unit tests illustrating using Future: `UsingCombineTests/FuturePublisherTests.swift`

Details

Future is a publisher that let's you combine in any asynchronous call and use that call to generate a value or a completion as a Publisher. It's ideal for when you want to make a single request, or get a single response, where the API you are using has a completion handler closure.

The obvious example that everyone immediately thinks about is URLSession. Fortunately, URLSession.dataTaskPublisher exists to make a call with a URLSession and return a publisher. However, if you already have an API object that wraps the direct calls to URLSession, then making a single request using Future can be a great way to integrate the result into a Combine pipeline.

There are a number of other APIs that exist in the Apple frameworks that use a completion closure. An example of one is requesting permission to access the contacts store in Contacts. An example of wrapping that request for access into a publisher using Future might be:

```
import Contacts
let futureAsyncPublisher = Future<Bool, Error> { promise in ①
    CNContactStore().requestAccess(for: .contacts) { grantedAccess, err in ②
        // err is an optional
        if let err = err { ③
            promise(.failure(err))
        }
        return promise(.success(grantedAccess)) ④
    }
}
```

① Future itself has you define the return types and takes a closure. It hands in a Result object matching the type description, which you interact.

② You can invoke the async API however is relevant, including passing in its required closure.

③ Within the completion handler, you determine what would cause a failure or a success. A call to `promise(.failure(<FailureType>))` returns the failure.

④ Or a call to `promise(.success(<OutputType>))` returns a value.

If you want to wrap an async API that could return many values over time, you should not use Future directly, as it only returns a single value. Instead, you should consider creating your own publisher based on passthroughSubject or currentValueSubject, or wrapping the Future publisher with Deferred.

Future creates and invokes its closure to do the asynchronous request **at the time of creation**, not when the publisher receives a demand request. This can be extremely counter-intuitive, as many other publishers invoke their closures when demand is placed to them. This also means that you can't directly link a Future publisher to an operator like retry.

The retry operator works by making another subscription to the publisher, and Future doesn't not currently re-invoke the closure you provide upon additional request demands. This means that chaining a retry() operator after future will not result in the Future's closure being invoked repeatedly when a `.failure` completion is returned.

The failure of the retry operator and Future to work together directly has been submitted to Apple as feedback: FB7455914.

The Future publisher can be wrapped with Deferred to have it work based on demand, rather than as a one-shot at the time of creation of the publisher. You can see unit tests illustrating Future wrapped with Deferred in the tests at `UsingCombineTests/FuturePublisherTests.swift`.

If you are wanting repeated requests to a Future (for example, wanting to use a a retry operator to retry failed requests), wrap the Future publisher with Deferred.

```
let deferredPublisher = Deferred { ①
    return Future<Bool, Error> { promise in ②
        self.asyncAPICall(sabotage: false) { (grantedAccess, err) in
            if let err = err {
                promise(.failure(err))
            }
            promise(.success(grantedAccess))
        }
    }
}.eraseToAnyPublisher()
```

① The closure provided in to Deferred will be invoked as demand requests come to the publisher.

② Which in turn resolves the underlying api call to generate the result as a Promise, with internal closures to resolve the promise.

Published

Summary

A property wrapper that adds a Combine publisher to any property

docs

Published

Usage

- Declarative UI updates from user input

- Cascading UI updates including a network request
- unit tests illustrating using Published: `UsingCombineTests/PublisherTests.swift`

Details

Published is part of Combine, but allows you to wrap a property, enabling you to get a publisher that triggers data updates whenever the property is changed. The publisher's output type is inferred from the type of the property, and the error type of the provided publisher is <Never>.

A smaller examples of how it can be used:

```
@Published var username: String = "" ①

$username ②
    .sink { someString in
        print("value of username updated to: ", someString)
    }

$username ③
    .assign(\.text, on: myLabel)

@Published private var githubUserData: [GithubAPIUser] = [] ④
```

① `@Published` wraps the property, username, and will generate events whenever the property is changed. If there is a subscriber at initialization time, the subscriber will also receive the initial value being set. The publisher for the property is available at the same scope, and with the same permissions, as the property itself.

② The publisher is accessible as `$username`, of type `Published<String>.publisher`.

③ A Published property can have more than one subscriber pipeline triggering from it.

④ If you're publishing your own type, you may find it convenient to publish an array of that type as the property, even if you only reference a single value. This allows you represent an "Empty" result that is still a concrete result within Combine pipelines, as assign and sink subscribers will only trigger updates on non-nil values.

If the publisher generated from `@Published` receives a cancellation from any subscriber, it is expected to, and will cease, reporting property changes. Because of this expectation, it is common to arrange pipelines from these publishers that have an error type of `<Never>` and do all error handling within the pipelines. For example, if a sink subscriber is set up to capture errors from a pipeline originating from a @Published property, when the error is received, the sink will send a `cancel` message, causing the publisher to cease generating any updates on change. This is illustrated in the test `testPublishedSinkWithError` at `UsingCombineTests/PublisherTests.swift`

Additional examples of how to arrange error handling for a continous publisher like `@Published` can be found at Using flatMap with catch to handle errors.

Using `@Published` should only be done within reference types - that is, within classes. An early beta (2) allowed @Published wrapped within a struct. As of beta5, the compiler will not throw an error if this is attempted:

```
<unknown>:0: error: 'wrappedValue' is unavailable: @Published is only
available on properties of classes
         Combine.Published:5:16: note: 'wrappedValue' has been
explicitly marked unavailable here
             public var wrappedValue: Value { get set }
                        ^
```

Empty

Summary

`empty` never publishes any values, and optionally finishes immediately.

docs

`Empty`

Usage

- Using catch to handle errors in a one-shot pipeline shows an example of using `catch` to handle errors with a one-shot publisher.
- Using flatMap with catch to handle errors shows an example of using `catch` with `flatMap` to handle errors with a continual publisher.
- Declarative UI updates from user input
- Cascading UI updates including a network request
- The unit tests at `UsingCombineTests/EmptyPublisherTests.swift`

Details

Empty is useful in error handling scenarios where with publishers where the value is an optional, or where you want to resolve an error by simply not sending anything. Empty can be invoked to be a publisher of any output and failure type combination.

Empty is most commonly used where you need to return a publisher, but don't want to propogate any values (a possible error handling scenario). If you want a publisher that provides a single value, then look at Just or Deferred publishers as alternatives.

When subscribed to, an instance of the Empty publisher will not return any values (or errors) and will immediately return a finished completion message to the subscriber.

An example of using Empty

```
let myEmptyPublisher = Empty<String, Never>() ①
```

① Because the types are not be able to be inferred, expect to always define the types you want to

return within the declaration.

Fail

Summary

`Fail` immediately terminates publishing with the specified failure.

docs

`Fail`

Usage

- The unit tests at `UsingCombineTests/FailedPublisherTests.swift`

Details

Fail is commonly used when implementing an API that returns a publisher. In the case where you want to return an immediate failure, Fail provides a publisher that immediately triggers a failure on subscription. One way this might be used is to provide a failure response when invalid parameters are passed. The Fail publisher lets you generate a publisher of the correct type that provides a failure completion when demand is requested.

Initializing a Fail publisher can be done two ways: with the type notation specifying the output and failure types or with the types implied by handing parameters to the initializer.

For example:

Initializing `Fail` by specifying the types

```
let cancellable = Fail<String, Error>(error: TestFailureCondition.exampleFailure)
```

Initializing `Fail` by providing types as parameters:

```
let cancellable = Fail(outputType: String.self, failure: TestFailureCondition
.exampleFailure)
```

Publishers.Sequence

Summary

Publishes a provided sequence of elements.

docs

`Publishers.Sequence`

Usage

- The unit tests at `UsingCombineTests/SequencePublisherTests.swift`

Details

Sequence provides a way to return values as subscribers demand them initialized from a

collection. Formally, it provides elements from any type conforming to the sequence protocol.

If a subscriber requests unlimited demand, all elements will be sent, and then a finished completion will terminate the output. If the subscribe requests a single element at a time, then individual elements will be returned based on demand.

If the type within the sequence is denoted as Optional, and a nil value is included within the sequence, that will be sent as an instance of the optional type.

Deferred

Summary

Publisher waits for a subscriber before running the provided closure to create values for the subscriber.

docs

`Deferred`

Usage

- The unit tests at `UsingCombineTests/DeferredPublisherTests.swift`
- The unit tests at `UsingCombineTests/FuturePublisherTests.swift`

Details

Deferred is useful when creating an API to return a publisher, where creating the publisher is an expensive effort, either computationally or in the time it takes to set up. Deferred holds off on setting up any publisher data structures until a subscription is requested. This provides a means of deferring the setup of the publisher until it's actually needed.

The Deferred publisher is particularly useful with Future, which does not wait on demand to start the resolution of underlying (wrapped) asynchronous APIs.

ObservableObjectPublisher

Summary

Used with SwiftUI, objects conforming to ObservableObject protocol can provide a publisher.

docs

`ObservableObjectPublisher`

Usage

- The unit tests at `UsingCombineTests/ObservableObjectPublisherTests.swift`

Details

When a class includes a Published property and conforms to the ObservableObject protocol, this class instances will get a `objectWillChange` publisher endpoint providing this publisher. The `objectWillChange` publisher will not return any of the changed data, only an indicator that the referenced object has changed.

The output type of `ObservableObject.Output` is type aliased to Void, so while it is not nill, it will not provide any meaningful data. Because the output type does not include what changes on the referenced object, the best method for responding to changes is probably best done using sink.

In practice, this method is most frequently used by the SwiftUI framework. SwiftUI views use the `@ObservedObject` property wrapper to know when to invalidate and refresh views that reference classes implementing ObservableObject.

Classes implementing ObservedObject are also expected to use @Published to provide notifications of changes on specific properties, or to optionally provide a custom announcement that indicates the object has changed.

It can also be used locally to watch for updates to a reference-type model.

SwiftUI

SwiftUI uses a variety of property wrappers within its Views to reference and display content from outside of those views. @Published, @ObservedObject, and @EnvironmentObject are the most common that also relate to Combine. SwiftUI also includes @Binding, which uses the Combine framework for similiar change notifications across SwiftUI views.

- @ObjectBinding (swiftUI)
- BindableObject
- often linked with method `didChange` to publish changes to model objects
    - `@ObjectBinding var model: MyModel`

Foundation

NotificationCenter

Summary

Foundation's NotificationCenter added the capability to act as a publisher, providing Notifications to pipelines.

Constraints on connected publisher

- *none*

docs

`NotificationCenter`

Usage

- Responding to updates from NotificationCenter
- The unit tests at `UsingCombineTests/NotificationCenterPublisherTests.swift`

Details

AppKit and MacOS applications have heavily relied on Notifications to provide general application state information. A number of components also use Notifications through NotificationCenter to provide updates on user interactions, such as

Notifications are identified primarily by name, defined by a string in your own code, or a constant from a relevant framework. You can find a good general list of existing Notifications by name at https://developer.apple.com/documentation/foundation/nsnotification/name. A number of framework specific notifications are often included within the framework. For example, within AppKit, there are a number of common notifications under NSControl.

A number of AppKit controls provide notifications when the control has been updated. For example, AppKit's TextField triggers a number of notifications including:

- textDidBeginEditingNotification
- textDidChangeNotification
- textDidEndEditingNotification

NotificationCenter provides a publisher upon which you may create pipelines to declaratively react to application or system notifications. When creating a publisher, you define a single Notification name, often from a constant within a relevant framework. The publisher optionally takes an object reference which further filters notifications to those provided by the specific reference.

```
extension Notification.Name {
    static let yourNotification = Notification.Name("your-notification") ①
}

let cancellable = NotificationCenter.default.publisher(for: .yourNotification, object:
nil) ②
    .sink {
        print ($0) ③
    }
```

① Notifications are defined by a string for their name. If defining your own, be careful to define the strings uniquely.

② A NotificationCenter publisher can be created for a single type of notification, `.yourNotification` in this case, defined previously in your code.

③ Notifications are received from the publisher. These include at least their name, and optionally a `object` reference from the sending object - most commonly provided from Apple frameworks. Notifications may also include a `userInfo` dictionary of arbitrary values, which can be used to pass additional information within your application.

Timer

Summary

Foundation's Timer added the capability to act as a publisher, providing Notifications to pipelines.

Constraints on connected publisher

- *none*

docs

`Timer`

Usage

- The unit tests at `UsingCombineTests/TimerPublisherTests.swift`

Details

Timer.publish returns an instance of `Timer.TimerPublisher`. This publisher is a connectable publisher, conforming to `ConnectablePublisher`. This means that even when subscribers are connected to it, it will not start producing values until connect() or autoconnect() is invoked on the publisher.

Creating the timer publisher requires an interval in seconds, and a RunLoop and mode upon which to run. The publisher may optionally take an additional parameter `tolerance`, which defines a variance allowed in the generation of timed events. The default for toleranace is nil, allowing any variance.

The publisher has an output type of Date and a failure type of `<Never>`.

If you want the publisher to automatically connect and start receiving values as soon as subscribers are connected and make requests for values, then you may include autoconnect() in the pipeline to have it automatically start to generate values as soon as a subscriber requests data.

```
let cancellable = Timer.publish(every: 1.0, on: RunLoop.main, in: .common)
    .autoconnect()
    .sink { receivedTimeStamp in
        print("passed through: ", receivedTimeStamp)
    }
```

Alternatively, you can connect up the subscribers, which will receive no values until you invoke `connect()` on the publisher, which also returns a Cancellable reference.

```
let timerPublisher = Timer.publish(every: 1.0, on: RunLoop.main, in: .default)
let cancellableSink = timerPublisher
    .sink { receivedTimeStamp in
        print("passed through: ", receivedTimeStamp)
    }
// no values until the following is invoked elsewhere/later:
let cancellablePublisher = timerPublisher.connect()
```

.publisher on KVO instance

Summary

Foundation added the ability to get a publisher on any Object that can be watched with Key Value Observing.

docs

'KeyValueObservingPublisher'

Usage

- The unit tests at `UsingCombineTests/PublisherTests.swift`

Details

Any Key Value Observing instance can produce a publisher. To create this publisher, you call the function `publisher` on the object, providing it with a single (required) KeyPath value.

For example:

```
private final class KVOAbleNSObject: NSObject {
    @objc dynamic var intValue: Int = 0
    @objc dynamic var boolValue: Bool = false
}

let foo = KVOAbleNSObject()

let _ = foo.publisher(for: \.intValue)
    .sink { someValue in
        print("value updated to: >>\(someValue)<<")
    }
```

KVO publisher access implies that with macOS 10.15 release or iOS 13, most of Appkit and UIKit interface instances will be accessible as publishers. Relying on the interface element's state to trigger updates into pipelines can lead to your state being very tightly bound to the interface elements, rather than your model. You may be better served by explicitly creating your own state to react to from a Published property wrapper.

URLSession.dataTaskPublisher

Summary

Foundation's `URLSession` has a publisher specifically for requesting data from URLs: dataTaskPublisher

Constraints on connected publisher

- *none*

docs

`URLSession.DataTaskPublisher`

Usage

- Making a network request with dataTaskPublisher
- Using catch to handle errors in a one-shot pipeline
- Retrying in the event of a temporary failure
- Requesting data from an alternate URL when the network is constrained
- Declarative UI updates from user input
- Cascading UI updates including a network request

Details

`dataTaskPublisher`, on URLSession, has two variants for creating a publisher. The first takes an instance of URL, the second URLRequest. The data returned from the publisher is a tuple of `(data: Data, response: URLResponse)`.

```
let request = URLRequest(url: regularURL)
return URLSession.shared.dataTaskPublisher(for: request)
```

RealityKit

- `RealityKit .Scene .publisher()`

Scene Publisher (from RealityKit)

- Scene.Publisher
  - SceneEvents
  - AnimationEvents
  - AudioEvents
  - CollisionEvents

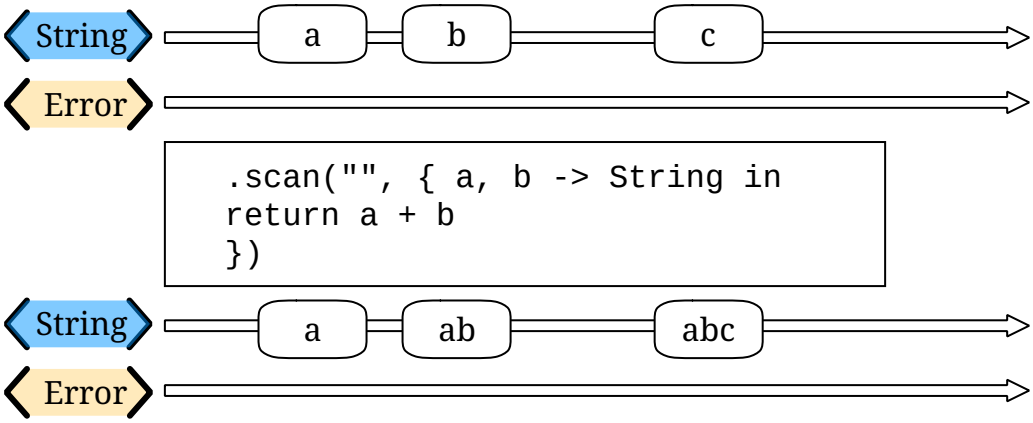
Operators

Mapping elements

scan

Summary

scan acts like an accumulator, collecting and modifying values according to a closure you provide, and publishing intermediate results with each change from upstream.



Constraints on connected publisher

- *none*

docs

https://developer.apple.com/documentation/combine/publishers/scan

While the published docs are unfortunately anemic, the generated swift headers has some detail:

```
/// Transforms elements from the upstream publisher by providing the current element
to a closure along with the last value returned by the closure.
///
///     let pub = (0...5)
///         .publisher
///         .scan(0, { return $0 + $1 })
///         .sink(receiveValue: { print ("\($0)", terminator: " ") })
///      // Prints "0 1 3 6 10 15 ".
///
///
/// - Parameters:
///   - initialResult: The previous result returned by the `nextPartialResult`
closure.
///   - nextPartialResult: A closure that takes as its arguments the previous value
returned by the closure and the next element emitted from the upstream publisher.
/// - Returns: A publisher that transforms elements by applying a closure that
receives its previous return value and the next element from the upstream publisher.
```

Usage

- unit tests illustrating using scan : `UsingCombineTests/ScanPublisherTests.swift`

Details

Scan lets you accumulate values or otherwise modify a type as changes flow through the pipeline. You can use this to collect values into an array, implement a counter, or any number of other interesting use cases.

tryScan

Summary

tryScan is a variant of the scan operator which allows for the provided closure to throw an error and cancel the pipeline. The closure provided updates and modifies a value with based on any inputs from an upstream publisher and publishing intermediate results.

Constraints on connected publisher

- *none*

docs

https://developer.apple.com/documentation/combine/publishers/tryscan

While the published docs are unfortunately anemic, the generated swift headers has some detail:

```
/// Transforms elements from the upstream publisher by providing the current element
to an error-throwing closure along with the last value returned by the closure.
///
/// If the closure throws an error, the publisher fails with the error.
/// - Parameters:
///   - initialResult: The previous result returned by the `nextPartialResult`
closure.
///   - nextPartialResult: An error-throwing closure that takes as its arguments the
previous value returned by the closure and the next element emitted from the upstream
publisher.
/// - Returns: A publisher that transforms elements by applying a closure that
receives its previous return value and the next element from the upstream publisher.
```

Usage

- unit tests illustrating using tryScan : `UsingCombineTests/ScanPublisherTests.swift`

Details

tryScan lets you accumulate values or otherwise modify a type as changes flow through the pipeline while also supporting an error state. If either the combined and updates values, or the incoming value, matches logic you define within the closure, you can throw an error, terminating the pipeline.

map

Summary

map is most commonly used to convert one data type into another along a pipeline.

Constraints on connected publisher

- *none*

docs

https://developer.apple.com/documentation/combine/publishers/map

n/a

Usage

- Making a network request with dataTaskPublisher
- Using catch to handle errors in a one-shot pipeline
- Retrying in the event of a temporary failure
- Declarative UI updates from user input
- Cascading UI updates including a network request
- unit tests illustrating using map with dataTaskPublisher: `UsingCombineTests/DataTaskPublisherTests.swift`

Details

The map operator doesn't allow for any additional failures to be thrown, and doesn't transform the failure type. If you want to throw an error within your closure, then use the tryMap operator.

map takes a single closure where you provide the logic for the map operation.

For example, the URLSession.dataTaskPublisher provides a tuple of `(data: Data, response: URLResponse)`` as its output. You can use map to pass along the data, for example to use with decode.

```
.map { $0.data } ①
```

① the `$0` indicates to grab the first parameter passed in, which is a tuple of `data` and `response`.

In some cases, the closure may not be able to infer what data type you are returning, so you may need to provide a definition to help the compiler. For example, if you have an object getting passed down that has a boolean property "isValid" on it, and you just want the boolean for your pipeline, you might set that up like:

```
struct MyStruct {
    isValid: bool = true
}
//
Just(MyStruct())
.map { inValue -> Bool in ①
  inValue.isValid ②
}
```

① inValue is named as the parameter coming in, and the return type is being explicitly specified to `Bool`

② A single line is an implicit return, in this case it's pulling the `isValid` property off the struct and passing it down the pipeline.

tryMap

Summary

tryMap is effectively the similiar to map, except that it also allows you to provide a closure that throws additional errors if your conversion logic is unsuccessful.

Constraints on connected publisher

- *none*

docs

https://developer.apple.com/documentation/combine/publishers/trymap

Usage

- Stricter request processing with dataTaskPublisher
- unit tests illustrating using tryMap with dataTaskPublisher: `UsingCombineTests/DataTaskPublisherTests.swift`

Details

tryMap is useful when you have more complex business logic around your map and you want to indicate that the data passed in is an error, possibly handling that error later in the pipeline. If you are looking at tryMap to decode JSON, you may want to consider using the decode operator instead, which is set up for that common task.

```
enum MyFailure: Error {
    case notBigEnough
}

//
Just(5)
.tryMap {
  if inValue < 5 { ①
      throw MyFailure.notBigEnough ②
  }
  return inValue ③
}
```

① You can specify whatever logic is relevant to your use case within tryMap

② and throw an error, although throwing an Error isn't required.

③ If the error condition doesn't occur, you do need to pass down data for any further subscribers.

flatMap

Summary

Used with error recovery or async operations that might fail (ex: Future), flatMap will replace any incoming values with another publisher.

Constraints on connected publisher

- *none*

docs

`flatMap`

Usage

- Using flatMap with catch to handle errors
- unit tests illustrating flatMap: `UsingCombineTests/SwitchAndFlatMapPublisherTests.swift`

Details

Most typically used in error handling scenarios, flatMap takes a closure that allows you to read the incoming data value, and provide a publisher that returns a value to the pipeline.

In error handling, this is most frequently used to take the incoming value and create a one-shot pipeline that does some potentially failing operation, and then handling the error condition with a catch operator.

A diagram version of this pipeline construct might be:

```
     one-shot-publisher(value) -> catch ( fallback )      // <- one-shot pipeline
                           ^                          \
                           |                           \
publisher -> flatMap -> ( +                             +  ) -> subscriber
```

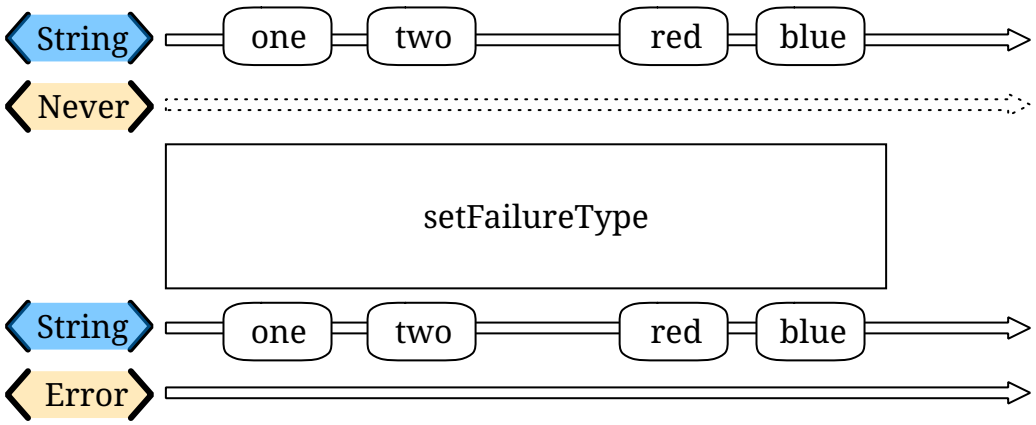
In Swift, this looks like:

```
.flatMap { data in
    return Just(data)
    .decode(YourType.self, JSONDecoder())
    .catch {
        return Just(YourType.placeholder)
    }
}
```

setFailureType

Summary

The publisher cannot actually fail with the specified type and instead just finishes normally. Use this publisher type when you need to match the error types for two otherwise mismatched publishers within a pipeline.



Constraints on connected publisher

- The upstream publisher must have a failure type of `<Never>`.

docs

`setFailureType`

Usage

- unit tests illustrating setFailureType: `UsingCombineTests/FailedPublisherTests.swift`

Details

setFailureType is an operator for transforming the error type within a pipeline, specifically from `<Never>` to some error type you may want to produce. setFailureType does not induce an error, but changes the types of the pipeline.

This can be especially convenient if you need to match an operator or subscriber that expects a failure type other than <Never> when you are working with a test or single-value publisher such as Just or Sequence.

Filtering elements

compactMap

- compactMap
  - republishes all non-nil results of calling a closure with each received element.
  - there's a variant `tryCompactMap` for use with a provided error-throwing closure.

tryCompactMap

- tryCompactMap

filter

Summary

Filter passes through all instances of the output type that match a provided closure, dropping any that don't match.

Constraints on connected publisher

- requires Failure type to be `<Never>`

docs

`filter`

Usage

- Declarative UI updates from user input
- Cascading UI updates including a network request
- unit tests illustrating using filter: `UsingCombineTests/FilterPublisherTests.swift`

Details

Filter takes a single closure as a parameter that is provided the value from the previous publisher and returns a Bool value. If the return from the closure is `true`, then the operator republishes the value further down the chain. If the return from the closure is `false`, then the operator drops the value.

If you need a variation of this that will generate an error condition in the pipeline to be handled use the tryFilter operator, which allows the closure to throw an error in the evaluation.

tryFilter

Summary

tryFilter passes through all instances of the output type that match a provided closure, dropping any that don't match, and allows generating an error during the evaluation of that closure.

Constraints on connected publisher

- none

docs

`tryFilter`

Usage

- unit tests illustrating using tryFilter: `UsingCombineTests/FilterPublisherTests.swift`

Details

Like filter, tryFilter takes a single closure as a parameter that is provided the value from the previous publisher and returns a Bool value. If the return from the closure is `true`, then the operator republishes the value further down the chain. If the return from the closure is `false`, then the operator drops the value. You can additionally throw an error during the evaluation of tryFilter, which will then be propogated as the failure type down the pipeline.

removeDuplicates

Summary

removeDuplicates remembers what was previously sent in the pipeline, and only passes forward values that don't match the current value.

Constraints on connected publisher

- Available when Output of the previous publisher conforms to Equatable.

docs

`removeDuplicates`

Usage

- unit tests illustrating using removeDuplicates: `UsingCombineTests/DebounceAndRemoveDuplicatesPublisherTests.swift`

Details

The default usage of removeDuplicates doesn't require any parameters, and the operator will publish only elements that don't match the previously sent element.

```
.removeDuplicates()
```

A second usage of removeDuplicates takes a single parameter `by` that accepts a closure that allows you to determine the logic of what will be removed. The parameter version does not have the constraint on the Output type being equatable, but requires you to provide the relevant logic. If the closure returns true, the removeDuplicates predicate will consider the values matched and not forward a the duplicate value.

```
.removeDuplicates(by: { first, second -> Bool in
    // your logic is required if the output type doesn't conform to equatable.
    first.id == second.id
})
```

A variation of removeDuplicates exists that allows the predicate closure to throw an Error exists: tryRemoveDuplicates

tryRemoveDuplicates

Summary

tryRemoveDuplicates is a variant of removeDuplicates that allows the predicate testing equality to throw an Error, resulting in an Error completion type.

Constraints on connected publisher

- none

docs

`tryRemoveDuplicates`

Usage

- unit tests illustrating using tryRemoveDuplicates: `UsingCombineTests/DebounceAndRemoveDuplicatesPublisherTests.swift`

Details

tryRemoveDuplicates is a variant of removeDuplicates taking a single parameter that can throw an error. The parameter is a closure that allows you to determine the logic of what will be removed. If the closure returns true, tryRemoveDuplicates will consider the values matched and not forward a the duplicate value. If the closure throws an error, a failure completion will be propogated down the chain, and no value is sent.

```
.removeDuplicates(by: { first, second -> Bool throws in
    // your logic is required if the output type doesn't conform to equatable.

})
```

replaceEmpty

- replaceEmpty
  - requires Failure to be `<Never>`

replaceError

- replaceError
  - requires Failure to be `<Never>`

replaceNil

- replaceNil
  - requires Failure to be `<Never>`
  - Replaces nil elements in the stream with the proviced element.

Reducing elements

collect

- collect
  - multiple variants
    - buffers items
    - `collect()` Collects all received elements, and emits a single array of the collection when the upstream publisher finishes.
    - `collect(Int)` collects N elements and emits as an array
    - `collect(.byTime)` or `collect(.byTimeOrCount)`

collectByCount

- collectByCount

collectByTime

- collectByTime

ignoreOutput

- ignoreOutput

reduce

- reduce
  - A publisher that applies a closure to all received elements and produces an accumulated value when the upstream publisher finishes.
  - requires Failure to be `<Never>`
  - there's a varient `tryReduce` for use with a provided error-throwing closure.

tryReduce

- tryReduce

Mathematic opertions on elements

max

- max
  - Available when Output conforms to Comparable.
  - Publishes the maximum value received from the upstream publisher, after it finishes.

min

- Publishes the minimum value received from the upstream publisher, after it finishes.
- Available when Output conforms to Comparable.

comparison

- comparison
  - republishes items from another publisher only if each new item is in increasing order from the previously-published item.
  - there's a variant `tryComparson` which fails if the ordering logic throws an error

tryComparison

- tryComparison

count

- count
  - publishes the number of items received from the upstream publisher

Applying matching criteria to elements

allSatisfy

- allSatisfy
  - Publishes a single Boolean value that indicates whether all received elements pass a given predicate.
  - there's a variant `tryAllSatisfy` when the predicate can throw errors

tryAllSatisfy

- tryAllSatisfy

contains

- contains
  - emits a Boolean value when a specified element is received from its upstream publisher.
  - variant `containsWhere` when a provided predicate is satisfied
  - variant `tryContainsWhere` when a provided predicate is satisfied but could throw errors

containsWhere

- containsWhere

tryContainsWhere

- tryContainsWhere

Applying sequence operations to elements

first

- first
  - requires Failure to be `<Never>`
  - publishes the first element to satisfy a provided predicate

firstWhere

- firstWhere

tryFirstWhere

- tryFirstWhere

last

- last
  - requires Failure to be `<Never>`
  - publishes the last element to satisfy a provided predicate

lastWhere

- lastWhere

tryLastWhere

- tryLastWhere

dropUntilOutput

- dropUntilOutput

dropWhile

- dropWhile

tryDropWhile

- tryDropWhile

concatenate

- concatenate

drop

- drop

    - multiple variants
    - requires Failure to be `<Never>`
    - Ignores elements from the upstream publisher until it receives an element from a second publisher.
    - or `drop(while: {})`

prefixUntilOutput

- prefixUntilOutput
    - Republishes elements until another publisher emits an element.
    - requires Failure to be `<Never>`

prefixWhile

- prefixWhile
    - Republishes elements until another publisher emits an element.
    - requires Failure to be `<Never>`

tryPrefixWhile

- tryPrefixWhile
    - Republishes elements until another publisher emits an element.
    - requires Failure to be `<Never>`

output

- output

Combining elements from multiple publishers

combineLatest

Summary

CombineLatest merges two pipelines into a single output, converting the output type to a tuple of values from the upstream pipelines, and providing an update when any of the upstream publishers provide a new value.

Constraints on connected publishers

- All upstream publishers must have the same failure type.

docs

- `combineLatest`
- `combineLatest3`
- `combineLatest4`

Usage

- Merging multiple pipelines to update UI elements
- unit tests illustrating using combineLatest: `UsingCombineTests/MergingPipelineTests.swift`

Details

CombineLatest, and its variants of combineLatest3 and combineLatest4, take multiple upstream publishers and create a single output stream, merging the streams together. CombineLatest merges two upstream publishers. ComineLatest3 merges three upstream publishers, and combineLatest4 merges four upstream publishers.

The output type of the operator is a tuple of the output types of each of the publishers. For example, if combineLatest was used to merge a publisher with the output type of `<String>` and another with the output type of `<Int>`, the resulting output type would be a tuple of `(<String>,<Int>)`.

CombineLatest is most often used with continual publishers, and it "remembers" the last output value provided from each publisher. In turn, when any of the upstream publishers sends an updated value, the operator makes a new combined tuple of all previous "current" values, adds in the new value in the correct place, and sends that new combined value down the pipeline.

The failure type of all three upstream publishers does need to be the same. For example, you can't have one publisher that has a failure type of Error and another (or more) that have a failure type of Never. If the combineLatest operator does receive a failure from any of the upstream publishers, then the operator (and the rest of the pipeline) is cancelled after propogating that failure.

If any of the upstream publishers finish normally (that is, they send a completion message of finished), the combineLatest operator will continue operating and processing any messages from any of the other publishers that has additional data to send.

Other operators that merge multiple upstream pipelines include merge and zip. If your upstream publishers have the same type and you want a stream of single values, as opposed to tuples, then you probably want to use the merge operator. If you want to wait on values from all upstream

provides before providing an updated value, then use the zip operator.

merge

Summary

Merge takes two upstream publishers and mixes the elements published into a single pipeline as they are received.

Constraints on connected publishers

- All upstream publishers must have the same output type.
- All upstream publishers must have the same failure type.

docs

- `merge`
- `merge3`
- `merge4`
- `merge5`
- `merge6`
- `merge7`
- `merge8`

Usage

- unit tests illustrating using merge: `UsingCombineTests/MergingPipelineTests.swift`

Details

Merge subscribers to two upstream publishers, and as they provide data for the subscriber it interleaves them into a single pipeline. Merge3 accepts three upstream publishers, merge4 accepts four upstream publishers, and so forth - through merge8 accepting eight upstream publishers.

In all cases, the upstreams publishers are required to have the same output type, as well as the same failure type.

As with combineLatest, if an error is propogated down any of the upstream publishers, the cancellation from the subscriber will terminate this operator and will propogate cancel to all upstream publishers as well.

If an upstream publisher completes with a normal finish, the merge operator continues interleaving and forwarding from any values other upstream publishers.

In the unlikely event that two values are provided at the same time from upstream publishers, the merge operator will interleave the values in the order upstream publishers are specified when the operator is initialized.

If you want to mix different upstream publisher types into a single stream, then you likely want to use either combineLatest or zip, depending on how you want the timing of values to be handled.

Other operators that merge multiple upstream pipelines include combineLatest and zip. If your upstream publishers have different types, but you want interleaved values to be propogated as they are available, use combineLatest. If you want to wait on values from all upstream provides before providing an updated value, then use the zip operator.

zip

Summary

Zip takes two upstream publishers and mixes the elements published into a single pipeline, waiting until values are paired up from each upstream publisher before forwarding the pair as a tuple.

Constraints on connected publishers

- All upstream publishers must have the same failure type.

docs

- `zip`
- `zip3`
- `zip4`

Usage

- unit tests illustrating using merge: `UsingCombineTests/MergingPipelineTests.swift`

Details

Zip works very similiarly to combineLatest, connecting 2 upstream publishers and providing the output of those publishers as a single pipeline with a tuple output type, composed of the types of the upstream publishers. Zip3 supports connecting three upstream publishers, and zip4 supports connecting four upstream publishers.

The notable difference from combineLatest is that zip will specifically wait for values to arrive from the upstream publishers, and will only publish a single new tuple when new values have been provided from all upstream publishers.

One example of using this is to wait until all streams have provided a single value to provide a synchronization point. For example, if you have 2 independent network requests and require them to both be complete before continuing to process the results, you can use zip to connect two URLSession.dataTaskPublisher, which will wait until both publishers are complete before forwarding the combined tuples.

Other operators that merge multiple upstream pipelines include combineLatest and merge. If your upstream publishers have different types, but you want interleaved values to be propogated as they are available, use combineLatest. If your upstream publishers have the same type and you want a stream of single values, as opposed to tuples, then you probably want to use the merge operator.

Handling errors

See Error Handling for more detail on how you can design error handling.

catch

Summary

The operator `catch` handles errors (completion messages of type `.failure`) from an upstream publisher by replacing the failed publisher with another publisher. The operator also transforms the Failure type to `<Never>`.

Constraints on connected publisher

- *none*

Documentation reference

`Publishers.Catch`

Usage

- Using catch to handle errors in a one-shot pipeline shows an example of using `catch` to handle errors with a one-shot publisher.
- Using flatMap with catch to handle errors shows an example of using `catch` with `flatMap` to handle errors with a continual publisher.
- Declarative UI updates from user input
- Cascading UI updates including a network request

Details

Once catch receives a `.failure` completion, it won't send any further incoming values from the original upstream publisher. You can also view catch as a switch that only toggles in one direction: to using a new publisher that you define, but only when the original publisher to which it is subscribed sends an error.

This can be illustrated with the following code snippet:

```
enum TestFailureCondition: Error {
    case invalidServerResponse
}

let simplePublisher = PassthroughSubject<String, Error>()

let _ = simplePublisher
    .catch { err in
        // must return a Publisher
        return Just("replacement value")
    }
    .sink(receiveCompletion: { fini in
        print(".sink() received the completion:", String(describing: fini))
    }, receiveValue: { stringValue in
        print(".sink() received \(stringValue)")
    })

simplePublisher.send("oneValue")
simplePublisher.send("twoValue")
simplePublisher.send(completion: Subscribers.Completion.failure(TestFailureCondition
.invalidServerResponse))
simplePublisher.send("redValue")
simplePublisher.send("blueValue")
simplePublisher.send(completion: .finished)
```

In this example, we are using a `PassthroughSubject` so that we can control when and what gets sent from the publisher. In the above code, we are sending two good values, then a failure, then attempting to send two more good values. The values you would see printed from our `.sink()` closures are:

```
.sink() received oneValue
.sink() received twoValue
.sink() received replacement value
.sink() received the completion: finished
```

When the failure was sent through the pipeline, catch intercepts it and returns "replacement value" as expected. The replacement publisher it used (`Just`) sends a single value and then sends a completion. If we want the pipeline to remain active, we need to change how we handle the errors.

tryCatch

Summary

A variant of the catch operator that also allows an `<Error>` failure type, and doesn't convert the failure type to `<Never>`.

Constraints on connected publisher

- *none*

docs

https://developer.apple.com/documentation/combine/publishers/trycatch

Usage

- Requesting data from an alternate URL when the network is constrained

Details

`tryCatch` is a variant of catch that has a failure type of `<Error>` rather than catch's failure type of `<Never>`. This allows it to be used where you want to immediately react to an error by creating another publisher that may also produce a failure type.

assertNoFailure

Summary

Raises a fatal error when its upstream publisher fails, and otherwise republishes all received input and converts failure type to `<Never>`.

Constraints on connected publisher

- *none*

docs

https://developer.apple.com/documentation/combine/publishers/assertnofailure

Usage

- Verifying a failure hasn't happened using assertNoFailure

Details

If you need to verify that no error has occured (treating the error output as an invariant), this is the operator to use. Like its namesakes, it will cause the program to terminate if the assert is violated.

Adding it into the pipeline requires no additional parameters, but you can include a string:

```
.assertNoFailure()
// OR
.assertNoFailure("What could possibly go wrong?")
```

I'm not entirely clear on where that string would appear if you did include it.

When trying out this code in unit tests, the tests invariably drop into a debugger at the assertion point when a .failure is processed through the pipeline.

If you want to convert an failure tyoe output of `<Error>` to `<Never>`, you probably want to look at the catch operator.

Apple asserts this function should be primarily used for testing and verifying "internal sanity checks that are active during testing".

retry

Summary

The retry operator is used to repeat requests to a previous publisher in the event of an error.

Constraints on connected publisher

- failure type must be `<Error>`

docs

https://developer.apple.com/documentation/combine/publishers/retry

Usage

- Retrying in the event of a temporary failure
- unit tests illustrating using map with dataTaskPublisher: `UsingCombineTests/DataTaskPublisherTests.swift`
- unit tests illustrating retry: `UsingCombineTests/RetryPublisherTests.swift`

Details

When you specify this operator in a pipeline and it receives a subscription, it first tries to request a subscription from its upstream publisher. If the response to that subscription fails, then it will retry the subscription to the same publisher.

The retry operator accepts a single parameter that specifies a number of retries to attempt.

Using retry with a high count can result in your pipeline not resolving any data or completions for quite a while, depending on how long each attempt takes. You may also want to consider also using the timeout operator to force a completion from the pipeline.

If the number of retries is specified and all requests fail, then the `.failure` completion is passed down to the subscriber of this operator.

In practice, this is mostly commonly desired when attempting to request network resources with an unstable connection. If you use a retry operator, you should add a specific number of retries so that the subscription doesn't effectively get into an infinite loop.

```
struct IPInfo: Codable {
    // matching the data structure returned from ip.jsontest.com
    var ip: String
}
let myURL = URL(string: "http://ip.jsontest.com")
// NOTE(heckj): you'll need to enable insecure downloads in your Info.plist for this
example
// since the URL scheme is 'http'

let remoteDataPublisher = URLSession.shared.dataTaskPublisher(for: myURL!)
    // the dataTaskPublisher output combination is (data: Data, response: URLResponse)
    .retry(3)
    // if the URLSession returns a .failure completion, retry at most 3 times to get a
successful response
    .map({ (inputTuple) -> Data in
        return inputTuple.data
    })
    .decode(type: IPInfo.self, decoder: JSONDecoder())
    .catch { err in
        return Publishers.Just(IPInfo(ip: "8.8.8.8"))
    }
    .eraseToAnyPublisher()
```

mapError

- mapError
  - Converts any failure from the upstream publisher into a new error.

Adapting publisher types

switchToLatest

Summary

A publisher that flattens any nested publishers, using the most recent provided publisher.

Constraints on connected publisher

- *none*

docs

'switchToLatest'

Usage

- Declarative UI updates from user input
- Cascading UI updates including a network request
- unit tests illustrating switchToLatest: UsingCombineTests/SwitchAndFlatMapPublisherTests.swift

Details

switchToLatest is akin to flatMap, taking in a publisher instance and returning its value (or values). The primary different is in where it gets the publisher. In flatMap, the publisher is returned within the closure provided to flatMap, and the operator works upon that to subscribe and provide the relevant value down the pipeline. In switchToLatest, the publisher instance is provided **as the output type** from a previous publisher or operator.

The most common form of using this is with a one-shot publisher such as Just getting its value as a result of a map transform.

It is also commonly used when working with an API that provides a publisher. switchToLatest assists in taking the result of the publisher and sending that down the pipeline rather than sending the publisher itself down as the output type.

The following snippet is part of the larger example Declarative UI updates from user input:

```
.map { username -> AnyPublisher<[GithubAPIUser], Never> in ②
    return GithubAPI.retrieveGithubUser(username: username) ①
}
// ^^ type returned in the pipeline is a Publisher, so we use
// switchToLatest to flatten the values out of that
// pipeline to return down the chain, rather than returning a
// publisher down the pipeline.
.switchToLatest() ③
```

① In this example, an API instance (GithubAPI) has a function that returns a publisher.

② We are using map to take an earlier String output type and use that to invoke the API, which returns a publisher instance.

③ We want to use the value from that publisher, not the publisher itself, which is exactly what switchToLatest() provides.

Controlling timing

debounce

Summary

debounce collapses multiple values within a specified time window into a single value

Constraints on connected publisher

- *none*

docs

'debounce'

Usage

- unit tests illustrating using debounce: `UsingCombineTests/DebounceAndRemoveDuplicatesPublisherTests.swift`

Details

The operator takes a minimum of two parameters, an amount of time over which to debounce the signal and a scheduler on which to apply the operations. The operator will collapse any values received within the timeframe provided to a single, last value received from the upstream publisher within the time window.

This operator is frequently used with removeDuplicates when the publishing source is bound to UI interactions, primarily to prevent an "edit and revert" style of interaction from triggering unnecessary work.

If you wish to control the value returned within the timewindow provided, you may prefer to use throttle, which allows you to choose the first or last value provided.

delay

Summary

Delays delivery of all output to the downstream receiver by a specified amount of time on a particular scheduler.

Constraints on connected publisher

- *none*

docs

'delay'

Usage

- Creating a repeating publisher by wrapping a delegate based API
- Retrying in the event of a temporary failure

Details

The delay operator passes through the data after a delay defined to the operator. The delay operator also requires a scheduler, where the delay is explicitly invoked.

```
.delay(for: 2.0, scheduler: headingBackgroundQueue)
```

measureInterval

- measureInterval
  - Measures and emits the time interval between events received from an upstream publisher.
  - requires Failure to be `<Never>`

throttle

Summary

Publishes either the most-recent or first element published by the upstream publisher in the specified time interval.

Constraints on connected publisher

- *none*

docs

'throttle'

Usage

- unit tests illustrating using throttle: `UsingCombineTests/DebounceAndRemoveDuplicatesPublisherTests.swift`

Details

Throttle is akin to the debounce operator in that it collapses values. The operator will collapse any values received within the timeframe provided to a single, last value received from the upstream publisher within the time window.

The operator takes a minimum of three parameters, `for`: an amount of time over which to collapse the values received, `scheduler`: a scheduler on which to apply the operations, and `latest`: a boolean indicating if the first value or last value should be chosen and forwarded.

This operator is frequently used with removeDuplicates when the publishing source is bound to UI interactions, primarily to prevent an "edit and revert" style of interaction from triggering unnecessary work.

```
.throttle(for: 0.5, scheduler: RunLoop.main, latest: false)
```

As of Xcode 11.2, the behavior for setting `latest` to false appears to have changed, as it's currently showing the same behavior as `true`. This has been reported to apple as Feedback FB7424221.

timeout

Summary

Terminates publishing if the upstream publisher exceeds the specified time interval without producing an element.

Constraints on connected publisher

- requires Failure to be `<Never>`

docs

https://developer.apple.com/documentation/combine/publishers/timeout

Usage

- unit tests illustrating using retry and timeout with dataTaskPublisher: `UsingCombineTests/DataTaskPublisherTests.swift`

Details

Timeout will force a resolution to a pipeline after a given amount of time, but does not guarantee either data or errors, only a completion. If a timeout does trigger and force a completion, it will not generate an failure completion with an error.

Timeout is specified with two parameters, a time period and a scheduler.

If you are using a specific background thread (for example, with the subscribe operator), then timeout should likely be using the same scheduler.

The time period specified will take a literal integer, but otherwise needs to conform to the protocol SchedulerTimeIntervalConvertible. If you want to set a number from a Float or Int, you need to create the relevant structure, as Int or Float directly doesn't conform. For example, if you're using a DispatchQueue, you could use DispatchQueue.SchedulerTimeType.Stride.

```
let remoteDataPublisher = urlSession.dataTaskPublisher(for: self.mockURL!)
    .delay(for: 2, scheduler: backgroundQueue)
    .retry(5) // 5 retries, 2 seconds each ~ 10 seconds for this to fall through
    .timeout(5, scheduler: backgroundQueue) // max time of 5 seconds before failing
    .tryMap { data, response -> Data in
        guard let httpResponse = response as? HTTPURLResponse,
            httpResponse.statusCode == 200 else {
                throw TestFailureCondition.invalidServerResponse
        }
        return data
    }
    .decode(type: PostmanEchoTimeStampCheckResponse.self, decoder: JSONDecoder())
    .subscribe(on: backgroundQueue)
    .eraseToAnyPublisher()
```

Encoding and decoding

encode

Summary

Encode converts the output from upstream Encodable object using a specified TopLevelEncoder. For example, use JSONEncoder or PropertyListEncoder..

Constraints on connected publisher

- Available when Output conforms to Encodable.

docs

https://developer.apple.com/documentation/combine/publishers/encode

Usage

- unit tests illustrating using encode and decode: `UsingCombineTests/EncodeDecodeTests.swift`

Details

The encode operator takes a single parameters:

- `encoder` an instance of an object conforming to TopLevelEncoder, frequently an instance of JSONEncoder() or PropertyListEncoder().

```
fileprivate struct PostmanEchoTimeStampCheckResponse: Codable {
    let valid: Bool
}

let dataProvider = PassthroughSubject<PostmanEchoTimeStampCheckResponse, Never>()
    .encode(encoder: JSONEncoder())
    .sink { data in
        print(".sink() data received \(data)")
        let stringRepresentation = String(data: data, encoding: .utf8)
        print(stringRepresentation)
    })
```

Like the decode operator, the encode process can also fail and throw an error, so it returns a failure type of Error. With the compiler forcing type matching, the usual error condition is if you flow an optional value into the pipeline.

decode

Summary

A very common operation is to want to use decode (or encode data in a pipeline, so Combine provides an operator specifically suited to that task.

Constraints on connected publisher

- Available when Output conforms to Decodable.

docs

https://developer.apple.com/documentation/combine/publishers/decode

Usage

- Making a network request with dataTaskPublisher
- Stricter request processing with dataTaskPublisher
- Using catch to handle errors in a one-shot pipeline
- Retrying in the event of a temporary failure
- unit tests illustrating using encode and decode: `UsingCombineTests/EncodeDecodeTests.swift`

Details

The decode operator takes two parameters:

- `type` which is typically a reference to a struct you've defined
- `decoder` an instance of an object conforming to TopLevelDecoder, frequently an instance of JSONDecoder() or PropertyListDecoder().

Since decoding can fail, the operator will also return a failure type of Error. The data type returned by the operator is defined by the type you provided to decode.

```
let testUrlString = "https://postman-echo.com/time/valid?timestamp=2016-10-10"
// checks the validity of a timestamp - this one should return {"valid":true}
// matching the data structure returned from https://postman-echo.com/time/valid
fileprivate struct PostmanEchoTimeStampCheckResponse: Decodable, Hashable {
    let valid: Bool
}

let remoteDataPublisher = URLSession.shared.dataTaskPublisher(for: URL(string:
testUrlString)!)
    // the dataTaskPublisher output combination is (data: Data, response: URLResponse)
    .map { $0.data }
    .decode(type: PostmanEchoTimeStampCheckResponse.self, decoder: JSONDecoder())
```

Working with multiple subscribers

multicast

- multicast

Debugging

breakpoint

Summary

A publisher that raises a debugger signal when a provided closure needs to stop the process in the debugger.

Constraints on connected publisher

- *none*

docs

https://developer.apple.com/documentation/combine/publishers/breakpoint

Usage

- Debugging pipelines with the debugger

Details

When any of the provided closures returns true, this publisher raises the SIGTRAP signal to stop the process in the debugger. Otherwise, this publisher passes through values and completions as-is.

The operator takes 3 optional closures as parameters, used to trigger when to raise the SIGTRAP signal:

- `receiveSubscription`
- `receiveOutput`
- `receiveCompletion`

```
.breakpoint(receiveSubscription: { subscription in
    return false // return true to throw SIGTRAP and invoke the debugger
}, receiveOutput: { value in
    return false // return true to throw SIGTRAP and invoke the debugger
}, receiveCompletion: { completion in
    return false // return true to throw SIGTRAP and invoke the debugger
})
```

breakpointOnError

Summary

Raises a debugger signal upon receiving a failure.

Constraints on connected publisher

- *none*

docs

https://developer.apple.com/documentation/combine/publishers/breakpoint/3205192-breakpointonerror

Usage

- Debugging pipelines with the debugger

Details

breakpointOnError is a convenience method used to raise a SIGTRAP signal when an error is propogated through it within a pipeline.

```
.breakpointOnError()
```

handleEvents

Summary

handleEvents is an all purpose operator that allow you to specify closures be invoked when publisher events occur.

Constraints on connected publisher

- *none*

docs

https://developer.apple.com/documentation/combine/publishers/handleevents

Usage

- unit tests illustrating using handleEvents: `UsingCombineTests/HandleEventsPublisherTests.swift`
- Debugging pipelines with the handleEvents operator

Details

handleEvents doesn't require any parameters, allowing you to specify what publisher events to which you'd like to respond. Optional closures can be provided for the following events:

- receiveSubscription
- receiveOutput
- receiveCompletion
- receiveCancel
- receiveRequest

All of the closures are expected to return Void, which makes handleEvents useful for intentionally creating side effects based on what is happening in the pipeline.

You could, for example, use handleEvents to update an activityIndicator UI element, triggering it on with the receipt of the subscription, and terminating with the receipt of either cancel or completion.

If you only want to view the information of what's happening, you might consider using the print operator instead.

```
.handleEvents(receiveSubscription: { _ in
    DispatchQueue.main.async {
        self.activityIndicator.startAnimating()
    }
}, receiveCompletion: { _ in
    DispatchQueue.main.async {
        self.activityIndicator.stopAnimating()
    }
}, receiveCancel: {
    DispatchQueue.main.async {
        self.activityIndicator.stopAnimating()
    }
})
```

print

Summary

Prints log messages for all publishing events.

Constraints on connected publisher

- *none*

docs

https://developer.apple.com/documentation/combine/publishers/print

Usage

- unit tests illustrating using print: `UsingCombineTests/PublisherTests.swift`
- Debugging pipelines with the print operator

Details

The print operator doesn't require a parameter, but if provided will preprend any console output with the string provided.

The print is incredibly useful to see "what's happening" within a pipeline, and can be used as "printf" debugging within the pipeline to see events.

Most of the example tests illustrating the operators within this reference use a print operator to provide additional text output within the tests to show what's happening.

The print operator isn't directly integrated with Apple's OSLog unified logging, although there is an optional `to` parameter that lets you specific an instance conforming to TextOutputStream to which

it will send the output.

```
let _ = foo.$username
    .print(self.debugDescription)
    .tryMap({ myValue -> String in
        if (myValue == "boom") {
            throw FailureCondition.selfDestruct
        }
        return "mappedValue"
    })
```

Scheduler and Thread handling operators

receive

Summary

Receive defines the scheduler on which to receive elements from the publisher.

Constraints on connected publisher

- *none*

docs

`receive`

Usage

- Creating a subscriber with assign shows an example of using assign to set an a boolean property on a UI element.
- unit tests illustrating using an assign subscriber in a pipeline from a dataTaskPublisher with subscribe and receive: `UsingCombineTests/SubscribeReceiveAssignTests.swift`

Details

Receive takes a single required parameter (`on:`) which accepts a scheduler, and an optional parameter (`optional:`) which can accept SchedulerOptions. Scheduler is a protocol in Combine, with the conforming types that are commonly used of RunLoop, DispatchQueue and OperationQueue. Receive is frequently used with assign to make sure any following pipeline invocations happen on a specific thread, such as `RunLoop.main` when updating user interface objects. Receive effects itself and any opertors chained after it, but not previous operators. If you want to influence a previously chained publishers (or operators) for where to run, you may want to look at the subscribe operator.

```
examplePublisher.receive(on: RunLoop.main)
```

Receive takes a single

subscribe

Summary

Subscribe defines the scheduler on which to run a publisher in a pipeline.

Constraints on connected publisher

- *none*

docs

`subscribe`

Usage

- Creating a subscriber with assign shows an example of using assign to set an a boolean property on a UI element.

- unit tests illustrating using an assign subscriber in a pipeline from a dataTaskPublisher with subscribe and receive: `UsingCombineTests/SubscribeReceiveAssignTests.swift`

Details

Subscribe assigns a scheduler to the preceding pipeline invocation. It is relatively infrequently used, specifically to encourage a publisher such as Just or Deferred to run on a specific queue. If you want to control which queue operators run on, then it is more common to use the receive operator, which effects all following operators and subscribers.

Subscribe takes a single required parameter (`on:`) which accepts a scheduler, and an optional parameter (`optional:`) which can accept SchedulerOptions. Scheduler is a protocol in Combine, with the conforming types that are commonly used of RunLoop, DispatchQueue and OperationQueue.

Subscribe effects a subset of the functions, and does not guarantee that a publisher will run on that queue. In particular, it effects a publishers `receive` function, the subscribers `request` function, and the `cancel` function. Some publishers (such as URLSession.dataTaskPublisher) have complex internals that will run on alterantive queues based on their configuration, and will be relatively uneffected by subscribe.

```
networkDataPublisher
    .subscribe(on: backgroundQueue) ①
    .receive(on: RunLoop.main) ②
    .assign(to: \.text, on: yourLabel) ③
```

① the `subscribe` call requests the publisher (and any pipeline invocations before this in a chain) be invoked on the backgroundQueue.

② the `receive` call transfers the data to the main runloop, suitable for updating user interface elements

③ the `assign` call uses the assign subscriber to update the property `text` on a KVO compliant object, in this case `yourLabel`.

When creating a DispatchQueue to use with Combine publishers on background threads, it is recommended that you use a regular serial queue rather than a concurrent queue to allow Combine to adhere to its contracts. That is - don't create the queue with `attributes: .concurrent`.

Type erasure operators

eraseToAnyPublisher

- when you chain operators together in Swift, the object's type signature accumulates all the various types, and it gets ugly pretty quickly.
- eraseToAnyPublisher takes the signature and "erases" the type back to the common type of AnyPublisher
- this provides a cleaner type for external declarations (framework was created prior to Swift 5's opaque types)
- `.eraseToAnyPublisher()`
- often at the end of chains of operators, and cleans up the type signature of the property getting assigned to the chain of operators

eraseToAnySubscriber

eraseToAnySubject

Subjects

General information on Subjects can be found in the Core Concepts section.

currentValueSubject

Summary

CurrentValue creates an object that can be used to integrate imperative code into a Combine pipeline, starting with an initial value.

docs

`CurrentValueSubject`

Usage

- Cascading UI updates including a network request

Details

currentValueSubject creates an instance to which you can attach multiple subscribers. When creating a currentValueSubject, you do so with an initial value of the relevant output type for the Subject.

CurrentValue remembers the current value so that when a subscriber is attached, it immediately receives the current value. When a subscriber is connected to it and requests data, the initial value is sent. Further calls to `.send()` afterwards will then send those values to any subscribers.

PassthroughSubject

Summary

PassthroughSubject creates an object that can be used to integrate imperative code into a Combine pipeline.

docs

`PassthroughSubject`

Usage

- Cascading UI updates including a network request

Details

PassthroughSubject creates an instance to which you can attach multiple subscribers. When it is created, only the types are defined.

When a subscriber is connected and requests data, it will not receive any values until a `.send()` call is invoked. Passthrough doesn't maintain any state, it only passes through provided values. Calls to `.send()` will then send values to any subscribers.

PassthroughSubject is commonly used in scenarios where you want to create a publisher from imperative code. One example of this might be a publisher from a delegate-callback structure, common in Apple's APIs. Another common use is to test subscribers and pipelines, providing you

with imperative control of when events are sent within a pipeline. When creating tests, you can send data (or a failure) is under test control.

Subscribers

For general information about subscribers and how they fit with publishers and operators, see Subscribers.

assign

Summary

Assign creates a subscriber used to update a property on a KVO compliant object.

Constraints on connected publisher

- Failure type must be `<Never>`

docs

`assign`

Usage

- Creating a subscriber with assign shows an example of using assign to set an a boolean property on a UI element.
- unit tests illustrating using an assign subscriber in a pipeline from a dataTaskPublisher with subscribe and receive: `UsingCombineTests/SubscribeReceiveAssignTests.swift`

Details

Assign only handles data, and expects all errors or failures to be handled in the pipeline before it is invoked. The return value from setting up assign can be cancelled, and is frequently used when disabling the pipeline, such as when a viewController is disabled or deallocated. Assign is frequently used in conjunction with the receive operator to receive values on a specific scheduler, typically `RunLoop.main` when updating UI objects.

The type of KeyPath required for the assign operator is important. It requires a ReferenceWritableKeyPath, which is different from both WritableKeyPath and KeyPath. In particular, ReferenceWritableKeyPath requires that the object you're writing to is a reference type (an instance of a class), as well as being publicly writable. A WritableKeyPath is one that's a mutable value reference (a mutable struct), and KeyPath reflects that the object is simply readable by keypath, but not mutable.

It's not always clear (for example, while using code-completion from the editor) what a property may reflect.

```
examplePublisher
    .receive(on: RunLoop.main) ②
    .assign(to: \.text, on: yourLabel) ③
```

An error you may see is

```
Cannot convert value of type 'KeyPath<SomeObject, Bool>' to specified
type 'ReferenceWritableKeyPath<SomeObject, Bool>'
```

This happens when you're attempting to assign to a property that is read-only. An example of this is UIActivityIndicator's `isAnimating` property.

Another error you might see on using the assign operator is:

```
Type of expression is ambiguous without more context
```

This error can occur when you are attempting to assign a non-optional type to a keypath that expects has an optional type. For example, UIImageView.image is of type `UIImage?`, so attempting to assign an output type of UIImage from a previous operator would result in this error message.

The solution is to either use sink, or to include a map operator prior to assignment that changes the output type to match. For example, to convert the type `UIImage` to `UIImage?` you could use:

```
.map { image -> UIImage? in
    image
}
```

sink

Summary

Sink creates an all-purpose subscriber. At a minimum, you provide a closure to receive values, and optionally a closure that receives completions.

Constraints on connected publisher

- *none*

docs

`sink`

Usage

- Creating a subscriber with sink shows an example of creating a sink that receives both completion messages as well as data from the publisher.
- unit tests illustrating a sink subscriber and how it works: `UsingCombineTests/SinkSubscriberTests.swift`

Details

There are two forms of the sink operator. The first is the simplest form, taking a single closure,

receiving only the values from the pipeline (if and when provided by the publisher). Using the simpler version comes with a constraint: the failure type of the pipeline must be `<Never>`. If you are working with a pipeline that has a failure type other than `<Never>`, you need to use the two closure version, or add error handling into the pipeline itself.

An example of the simple form of sink:

```
let examplePublisher = Just(5)

let cancellable = examplePublisher.sink { value in
    print(".sink() received \(String(describing: value))")
}
```

Be aware that the closure may be called repeatedly. How often it is called depends on the pipeline to which it is subscribing. The closure you provide is invoked for every update that the publisher passes down, up until the completion, and prior to any cancellation.

It may be tempting to ignore the cancellable you get returned from sink. For example, the code:

```
let _ = examplePublisher.sink { value in
    print(".sink() received \(String(describing: value))")
}
```

However, this has the side effect that as soon as the function returns, the ignored variable is deallocated, causing the pipeline to be cancelled. If you want the pipeline to operate beyond the scope of the function (you probably do), then assign it to a longer lived variable that doesn't get deallocated until much later. Simply including a variable declaration in the enclosing object is often a good solution.

The second form of sink takes two closures, the first of which receives the data from the pipeline, and the second receives pipeline completion messages. te a sink with two closures. The closures parameters are `receiveCompletion` and `receiveValue`: The .failure completion may also encapsulate an error.

An example of the two-closure sink:

```
let examplePublisher = Just(5)

let cancellable = examplePublisher.sink(receiveCompletion: { err in
    print(".sink() received the completion", String(describing: err))
}, receiveValue: { value in
    print(".sink() received \(String(describing: value))")
})
```

The type that is passed into receiveCompletion is the enum `Subscribers.Completion`. The completion

`.failure` incudes an Error wrapped within it, providing access to the underlying cause of the failure. To get to the error within the `.failure` completion, `switch` on the returned completion to determine if it is `.finished` or `.failure`, and then pull out the error.

When you chain a `.sink` subscriber onto a publisher (or pipeline), the result is cancellable. At any time before the publisher sends a completion, the subscriber can send a cancellation and invalidate the pipeline. After a cancel is sent, no further values will be received by either closure in the sink.

```
let simplePublisher = PassthroughSubject<String, Never>()
let cancellablePipeline = simplePublisher.sink { data in
  // do what you need with the data...
}

cancellablePublisher.cancel() // when invoked, this invalidates the pipeline
// no further data will be received by the sink
```

AnyCancellable is often used with the result of sink to convert the resulting type into AnyCancellable.

AnyCancellable

Summary

AnyCancellable type erases a subscriber to the general form of Cancellable.

docs

https://developer.apple.com/documentation/combine/anycancellable

Usage

- Declarative UI updates from user input
- Cascading UI updates including a network request
- Creating a repeating publisher by wrapping a delegate based API

Details

This is used to provide a reference to a subscriber that allows the use of `cancel` without access to the subscription itself to request items. This is most typically used when you want a reference to a subscriber to clean it up on deallocation. Since the assign returns an AnyCancellable, this is often used when you want to save the reference to a sink an AnyCancellable.

```
var mySubscriber: AnyCancellable?

let mySinkSubscriber = remotePublisher
    .sink { data in
        print("received ", data)
    }
mySubscriber = AnyCancellable(mySinkSubscriber)
```