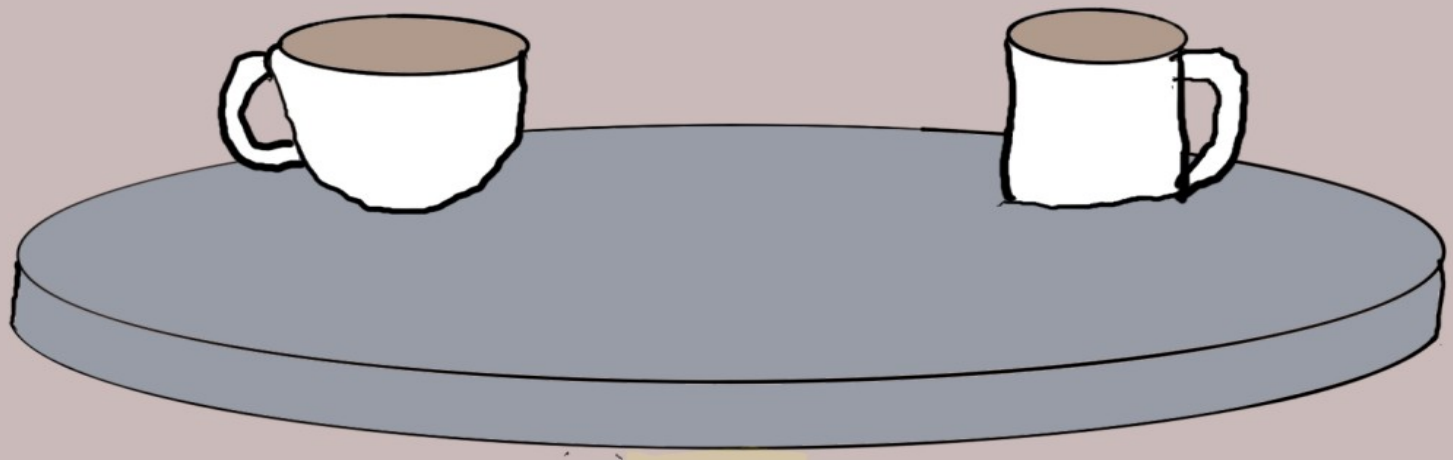
The Curious Case of the Async Cafe

An Introduction to
Modern Concurrency in Swift



by Daniel H Steinberg

The Curious Case Of The Async Cafe

An Introduction To

Modern Concurrency In Swift

by Daniel H Steinberg

Editors Cut

Copyright

ISBN-13: 978-1-944994-04-4

Book Version

This is version 0.5 for Swift 5.7, Xcode 14.2, macOS Ventura 13.0, and iOS 16.1 released February 2023. All code has been tested on Apple Silicon.

Code Download

Visit https://github.com/editorscut/ec013Async for all of the code for this book.

Run it in Xcode 14.2 or higher. All code is written in Swift.

To avoid long lines and code that wraps, I've split some lines in code listings in ways that you might not in an IDE. Please feel free to not break the lines where I have.

Recommended Settings

The ePub is best viewed in scrolling mode on an iPad. On smaller devices I also choose landscape. For some reason that I don't understand, scrolling mode is supported by Apple's Books app on

the iPad but not on the Mac. If you view this book in Apple's Books app, choose "Let lines break naturally" in Preferences > General. Finally, I've gone to great pains to make this look good in light and dark mode but Apple has foiled me yet again. I'm told that not all of the syntax coloring works in dark mode.

Submit Errata

Submit your errata here for the book or for the source code by selecting New Issue. Please provide the book version listed above, chapter, section, and page number in your issue so that I can find it and, if possible, resolve it quickly.

Official Links

Please check http://developer.apple.com for additional resources including videos, sample code, documentation, and forums. You'll also find information on what is required to take advantage of these resources.

Apple has posted videos, slides, and sample code from the Worldwide Developers Conference.

Legal

The sample code is intended to be used to illustrate points made in the text. It is not intended to be used in production code.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks or service marks. Where those designations appear in this book, and Dim Sum Thinking, Inc. was aware of the trademark claim, the designations have been printed with initial capital letters or in all capitals.

This book uses terms that are registered trademarks of Apple Inc. for which the terms of use don't permit rendering them in all caps or initial caps. You can view a complete list of the trademarks and registered trademarks of Apple Inc at http://www.apple.com/legal/trademark/appletmlist.html.

The Editor's Cut name and logo are registered trademarks of Dim Sum Thinking, Inc.

Table Of Contents

CHAPTER 3

AsyncSequences And AsyncStreams

Sections:

- Notifications
- Introducing AsyncStream
- Sequences of Notifications
- Sendable and Actor Boundaries
- Transforming AsyncSequences
- Sequence Pipelines
- Combine
- AsyncStream Continuations
- Continuous Delivery
- AsyncAlgorithms

We ended the previous chapter with an example of a `URLSession` data task. This is an asynchronous call that can fail and only returns once. We start this chapter with the example of `NotificationCenter`. We register to listen for a notification and can get zero or many calls over time.

This is our introduction to `AsyncSequence`s and an easy to use concrete implementation called an `AsyncStream`.

We iterate these sequences using an asynchronous version of a `for` loop in which we `await` the next element in the asynchronous sequence.

We construct and use these in many different ways and use built in methods such as `filter()` and `map()` to transform the `AsyncSequence`. We finish the chapter with a Swift Package that is outside of the Swift Standard Library named `AsyncAlgorithms` that is filled with types and functions that allow us to combine `AsyncSequence`s and work with them in other important ways.

We begin with a familiar look at `Notification`s and `NotificationCenter`.

AsyncAlgorithms

"Edges," I hissed, gesturing, "the magician is back."

Edges nodded and we watched as the magician placed two steaming cups of coffee in front of us.

The steam turned to smoke. When the smoke cleared the cups were gone and each of our napkins was half light yellow and half pale purple.

The miniature train appeared beside our table with an apple fritter on the light yellow car and a chocolate eclair on the pale purple car.

The magician took out four domes and covered each of the pastries and each of our napkins.

Edges understood that the trick was already complete and looked to the magician for permission. The magician nodded and as the train returned to the kitchen, Edges lifted the domes in front of each of us to reveal that we each had a plate with half an apple fritter and half an eclair.

A train with an endless supply of pastries and a magician who can transform them.

It can't get any better than this, I thought.

As if reading my mind, Edges said softly, "Enjoy, my friend. This is the end of endless pastry at the Async Cafe."

The Swift team has made an interesting choice. `AsyncSequence`, `AsyncStream`, and others have been added as part of the Swift Standard Library. `AsyncSequence` includes ways of transforming the sequence including `dropFirst()`, `filter()`, `map()`, and so on.

There are, however, many things missing that we might want. There are time based operations such as `debounce()` and `throttle()`. There are methods for combining asynchronous sequences that we might want to see such as `zip()`, `merge()`, or `combineLatest()`. There are also utilities that we might be missing such as `compacted()`, `removeDuplicates()`, and `interspersed()`.

All of these methods and more are contained in a Swift Package named *AsyncAlgorithms*.

In this section we explore some of the methods in `AsyncAlgorithms`.

Continue with our current project or start with the project in *Chapter03/09/*.

The set up

Let's do a little cleanup and preparation.

Remove the second `EntryGrid` from `MainView`.

*SmoreNmore/Views/**MainView.swift***

```
extension MainView: View {
  var body: some View {
    NavigationStack {
      VStack {
        EntryGrid(entries: controller.entries)
        EntryGrid(entries: controller.entries2)
        // EntryPairGrid(entryPairs: controller.entryPairs)
      }
      .padding()
      .navigationTitle("Entries")
      .navigationBarTitleDisplayMode(.inline)
    }
  }
}
```

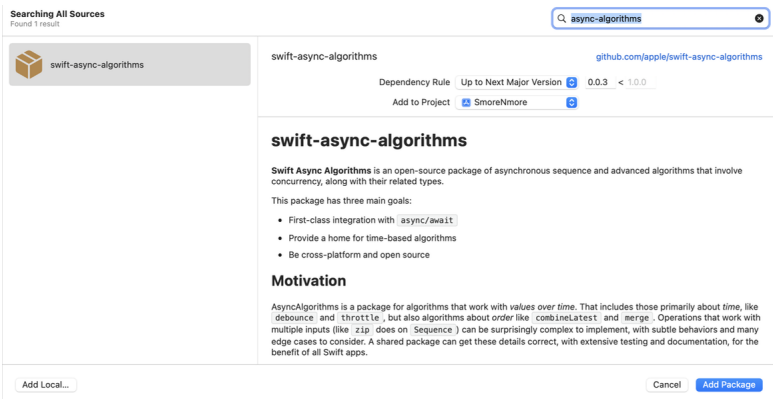
Remove `entries2` and `listenForEntries2()` from `EntryController`.

*SmoreNmore/Controllers/**EntryController.swift***

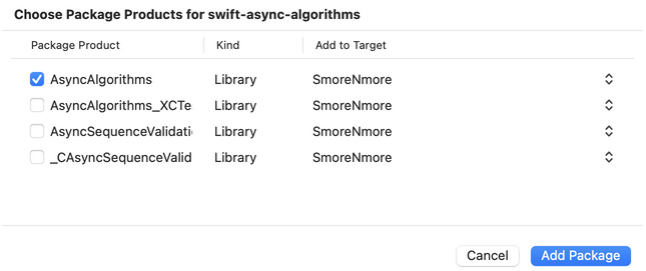
```
import Combine

@MainActor
class EntryController: ObservableObject {
  @Published private(set) var entries: [Entry] = []
  @Published private(set) var entries2: [Entry] = []
  @Published private(set) var entryPairs: [EntryPair] = []

  private let plain = AutoEntryVendor(delay: 2.0)
  private let filled = AutoEntryVendor(delay: 1.5,
                                       isFilled: true)

  init() {
    Task {
      await listenForEntries()
    }
    Task {
      await listenForEntries2()
    }
  }
}

extension EntryController {
  private func listenForEntries() async {
    for await entry in plain.entries {
      entries.append(entry)
    }
  }
  private func listenForEntries2() async {
    for await entry in filled.entries {
      entries2.append(entry)
    }
  }
}
```

Add AsyncAlgorithms

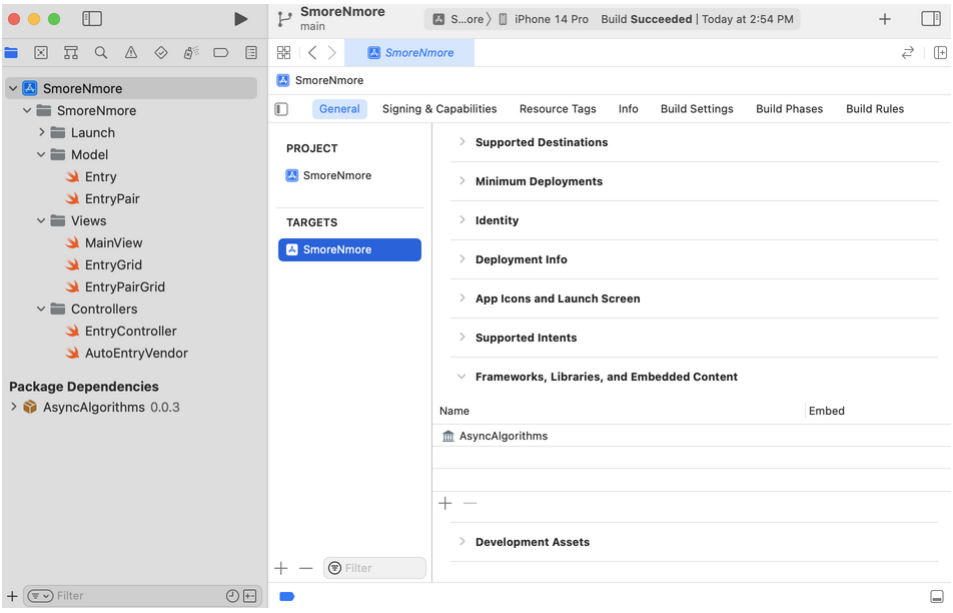
To add the `AsyncAlgorithms` package choose the menu item File > Add Packages. Search using the term async-algorithms or use the url `https://github.com/apple/swift-async-algorithms`. Tap Add Package.



After a moment you will see this window which asks you which of the four packages you want to add. Check the top one *AsyncAlgorithms* and again tap Add Package.



To confirm that the package has been added you should see it listed under *Package Dependencies* in the Project Navigator. Also if you select the target and look at the `General` tab, you should see it listed under *Frameworks, Libraries, and Embedded Content*.



This is important. Every once in a while there's an issue with adding the package and it doesn't show up in *Frameworks, Libraries, and Embedded Content*. If it doesn't, then tap the + for that section and add the package to the project manually.

Run the app. The first ten numbers should appear in plain `Entry`s as before.

AsyncTimerSequence

Let's refactor our `AsyncStream` in `AutoEntryVendor` using `AsyncTimerSequence` from the `AsyncAlgorithms` package.

We can create an `AsyncTimerSequence` that fires after a specified duration. So we can use it to fire every two seconds instead of sleeping a `Task` for two seconds in a `while` loop.

We can still `finish()` the `Task` when `count` reaches some specified amount. Let's increase that amount to 20.

Here's the modified `entries` stream.

*SmoreNmore/Controllers/**AutoEntryVendor.swift***

```
import AsyncAlgorithms

class AutoEntryVendor {
  let delay: Double
  let isFilled: Bool
  private var count = 0

  init(delay: Double,
       isFilled: Bool = false) {
    self.delay = delay
    self.isFilled = isFilled
  }

  lazy private(set) var entries
  = AsyncStream(Entry.self) { continuation in
    let timer
    = AsyncTimerSequence.repeating(every: .seconds(delay))

    Task {
      for await _ in timer {
        if count < 20 {
          count += 1
          try? await Task.sleep(for: .seconds(delay))
          continuation.yield(Entry(number: count,
                                   isFilled: isFilled))
        } else {
          continuation.finish()
        }
      }
    }
  }
}
```

Run the app. It runs as before except this time the numbers go up to 20.

Merge

If we have two sequences in which the type of elements are the same, we can merge them into a single sequence using the `merge()` function from `AsyncAlgorithms`.

As you saw earlier in this chapter, the type of the resulting sequence is not so simple. We'd like to merge two `AsyncSequence`s of `Entry`s and get an `AsyncSequence` of `Entry`s. The result of `merge()` is an `AsyncMerge2Sequence` that is generic in the two types of sequences it merges.

Oh one more note before we take it for a spin. Note that it's an `AsyncMerge2Sequence`. There is a variation of `merge()` that accepts three sequences and produces an `AsyncMerge3Sequence`.

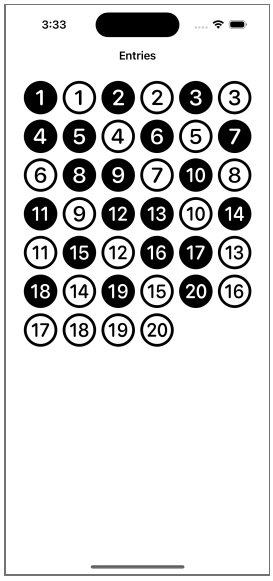
Let's use `merge()` to merge the elements from `plain.entries` and `filled.entries`.

*SmoreNmore/Controllers/**EntryController.swift***

```
import Combine
import AsyncAlgorithms
// ...
extension EntryController {
  private func listenForEntries() async {
    for await entry in merge(plain.entries,
                             filled.entries) {
      entries.append(entry)
    }
  }
}
```

Run the app. The `filled.entries` are emitted every 1.5 seconds while the `plain.entries` arrive every 2 seconds. We should see a filled `Entry` first followed by a plain one. They will alternate until the times line up so that we get two filled `Entry`s before the next plain `Entry` arrives.

I see something like this in the simulator.



The filled ones are exhausted first so at the end we get the remaining plain `Entry`s.

Let's mess around with the timing a little more.

Throttle

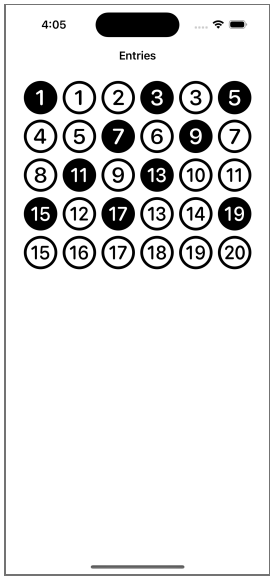
Sometimes we need to adjust the rate at which elements are being emitted by a sequence so we don't get an "I Love Lucy" chocolate factory situation (look it up if you don't know the iconic episode from more than 70 years ago).

The easiest way to see what happens is to modify our code and rerun our app. Let's throttle the faster `AsyncStream`. `filled.entries` sends a new `Entry` every 1.5 seconds. Let's set a throttle to 2 seconds like this.

*SmoreNmore/Controllers/**EntryController.swift***

```
extension EntryController {
  private func listenForEntries() async {
    for await entry in merge(plain.entries,
                             filled.entries
                               .throttle(for: .seconds(2.0))) {
      entries.append(entry)
    }
  }
}
```

Did you get the results you expected?



The `filled.entries` serves up a new `Entry` every 1.5 seconds but we used `throttle()` to say, "I need two seconds to recover from the previous one."

So after filled 1 we need a break of two seconds. Meanwhile, filled 2 is emitted 1.5 seconds after filled 1.

There's no buffering so we don't get told about filled 2. We miss out on it completely. A half second after filled 2 is created we're ready for another one so in another second we get filled 3.

We'll look at a couple more `AsyncAlgorithms` and consider some preconceptions we have.

Zip

Like many of the `AsyncAlgorithms`, `zip()` takes its inspiration from ordinary `Sequence`s. If we zip the two arrays `[a,b,c]` and `[1, 2]` together we get a `Sequence` that is something like this array of pairs `[(a, 1), (b, 2)]`. We pair up the elements into a new `Sequence` until we exhaust either sequence.

So, how should that work with `AsyncSequence`s? It's the same but we may have to wait for the corresponding entry from the other `AsyncSequence`.

We've got some setup work to do.

Uncomment the `EntryPairGrid` in `MainView`.

*SmoreNmore/Views/**MainView.swift***

```
extension MainView: View {
  var body: some View {
    NavigationStack {
      VStack {
        EntryGrid(entries: controller.entries)
        EntryPairGrid(entryPairs: controller.entryPairs)
      }
      .padding()
      .navigationTitle("Entries")
      .navigationBarTitleDisplayMode(.inline)
    }
  }
}
```

Add `listenForEntryPairs()` to `EntryController`. It will iterate over the `zip()` of `plain.entries` and `filled.entries` and modify `entryPairs`. We'll also simplify `listenForEntries()` to just use `plain.entries` again.

*SmoreNmore/Controllers/**EntryController.swift***

```
extension EntryController {
  private func listenForEntries() async {
    for await entry in plain.entries {
      entries.append(entry)
    }
  }
  private func listenForEntryPairs() async {
    for await pair in zip(plain.entries,
                          filled.entries) {
      entryPairs.append(EntryPair(pair))
    }
  }
}
```

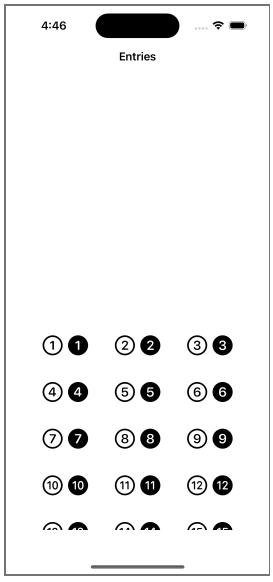
As a sanity check, run the app. The plain entries should appear on the screen as before.

To check out the `zip()` change the `init()` to call `listenForEntryPairs()`.

*SmoreNmore/Controllers/**EntryController.swift***

```
init() {
  Task {
    await listenForEntryPairs()
  }
}
```

The `zip()` runs as we expect. We see each plain entry paired with the corresponding filled entry even though the filled ones come out more quickly.



Before moving on, I want to show you what might be unexpected behavior - particularly if you're used to working with Combine publishers.

One and only one

With Combine we can arrange that more than one subscriber is connected to the same publisher. Every time the publisher publishes a new value, each subscriber will get it.

That's not the case with `AsyncSequences`. At least for now, if two `for await in` loops are listening to the same `AsyncSequence`, elements from the sequence will go to one or the other but not both.

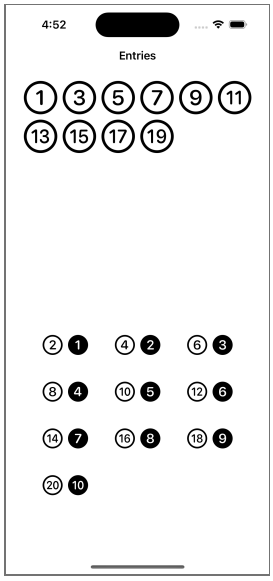
Here's a quick demonstration of that.

Add a `Task` with the work `listenForEntries()` to `init()`.

*SmoreNmore/Controllers/**EntryController.swift***

```
init() {
  Task {
    await listenForEntries()
  }
  Task {
    await listenForEntryPairs()
  }
}
```

Run the app. The plain entries are split between the top grid and the bottom pairs grid. This also means that when we run out of plain entries, the pairs finish.



We can see the plain and pairs finishing by adding this `onTermination` to `entries`.

*SmoreNmore/Controllers/**AutoEntryVendor.swift***

```
lazy private(set) var entries
= AsyncStream(Entry.self) { continuation in
  let timer
  = AsyncTimerSequence.repeating(every: .seconds(delay))
  continuation.onTermination = { termination in
    print("Stopped (is filled =", self.isFilled, ")"
          , termination)
  }
  Task {
    for await _ in timer {
      if count < 20 {
        count += 1
        continuation.yield(Entry(number: count,
                                 isFilled: isFilled))
      } else {
        continuation.finish()
      }
    }
  }
}
```

Run the app and we see this in the Console.

```
Stopped (is filled = false ) finished
Stopped (is filled = false ) finished
Stopped (is filled = true ) cancelled
```

The first two come from the plain one finishing. One report comes from the `listenForEntries()` method and the other comes from the `listenForEntryPairs()` method. Finally, when the plain entries are finished, `zip()` knows it's complete and cancels listening for the filled entries even though we've only used ten of them.

CombineLatest

Let's look at one more example as sometimes we want to reevaluate a pair of entries when either `AsyncSequence` updates.

For example, an `AsyncSequence` is a great way to listen for changes to a subscription or other authorization and we may want to take some action only when two different authorizations are valid.

In our simple example, delete the `EntryGrid` in `MainView` so that more of the results appear on the screen.

*SmoreNmore/Views/**MainView.swift***

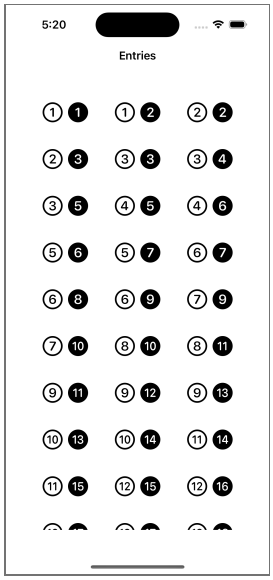
```
extension MainView: View {
  var body: some View {
    NavigationStack {
      VStack {
        ~~EntryGrid(entries: controller.entries)~~
        EntryPairGrid(entryPairs: controller.entryPairs)
      }
      .padding()
      .navigationTitle("Entries")
      .navigationBarTitleDisplayMode(.inline)
    }
  }
}
```

Remove `listenForEntries()` in `EntryController` and change `zip()` to `combineLatest()`.

*SmoreNmore/Controllers/**EntryController.swift***

```
import Combine

@MainActor
class EntryController: ObservableObject {
  @Published private(set) var entries: [Entry] = []
  @Published private(set) var entryPairs: [EntryPair] = []

  private let plain = AutoEntryVendor(delay: 2.0)
  private let filled = AutoEntryVendor(delay: 1.5,
                                       isFilled: true)

  init() {
    Task {
      await listenForEntries()
    }
    Task {
      await listenForEntryPairs()
    }
  }
}

import AsyncAlgorithms

extension EntryController {
  private func listenForEntries() async {
    for await entry in plain.entries {
      entries.append(entry)
    }
  }
  private func listenForEntryPairs() async {
    for await pair in combineLatest(plain.entries,
                                    filled.entries) {
      entryPairs.append(EntryPair(pair))
    }
  }
}
```

Run the app. This time you can see a pair whenever either the plain or the filled changes.



The final thing to warn you about `AsyncAlgorithms` is that the latest releases are initially tied to the current Swift toolchain so it will be harder to use this package if you aren't keeping your tools current.

That completes our look at `AsyncSequence`s and `AsyncStream`s. In the next chapter we explore structured concurrency.