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

"The Curious Case of the Async Cafe", by Daniel H Steinberg

Copyright © 2023 Dim Sum Thinking, Inc. All rights reserved.

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

Every precaution was taken in the preparation of this book. The publisher and author assume no responsibility for errors and omissions, or for damages resulting from the use of the information contained herein and in the accompanying code downloads.

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 4

Structured Concurrency

So far we've started with various situations and introduce various methods or types that are part of the async / await ecosystem to make things better.

In this chapter we look at examples of when this isn't enough. We want more control over `Task`s and may want to coordinate them.

This is where structured concurrency comes in.

We'll build a simple example using what we've learned so far to make asynchronous calls to create two `Entry`s and compare them. We then use `async let` so that the random numbers for both `Entry`s can be calculated at the same time. Part of this example will allow us to dig in a bit deeper into `Task` creation and cancellation.

That's great for two values, but what if we have a whole bunch of values?

We use `TaskGroup`s to work with many asynchronous calls and handle each as it is completed.

This is one of those chapters where you're going to keep saying, "that's just what I was looking for."

TaskGroups

Edges was already seated at an outside table at the Async Cafe when I arrived.

"I would suggest pastry again today," I said, "but poor Wiggins looked exhausted by the end yesterday."

Edges nodded, "I noticed that too. But don't worry. Today Wiggins has ensured that more of the Irregulars are available. Go ahead and make your list."

I handed my list to Edges who glanced at it and gave a loud whistle.

For the next few minutes one young person after another arrived, was given a pastry order, and then ran off to fulfill it.

Over the next twenty minutes of so, the youths returned. Some had had to travel further to fetch their assigned pastry and some were just faster than others.

As each returned, Edges thanked them, took the pastry, checked it off of the list, divided it and placed half in front of each of us.

"This," said Edges, "was a task intended for a group. A group such as the Baker's Treat Irregulars."

`async let` is a great device for executing a small number of asynchronous tasks concurrently when the order in which they will be used is known.

Sometimes, however, we have a large, dynamic, or unknown number of asynchronous tasks we wish to process concurrently and we're happy to use them in the order the tasks complete.

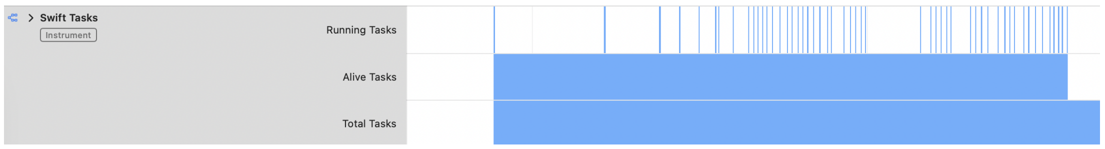
`TaskGroup`s are the mechanism designed to help us with exactly this situation. We'll start working with them in this section.

Continue with our current project or start with the project in *Chapter04/05/*.

The "Before" picture

Before we refactor our app using structured concurrency, let's take a snapshot by running the app using the Swift Concurrency instrument.

I entered a search term and see this for alive and running tasks.



When I enter the search a `Task` is created. You can see this is the only `Task` alive during the search. It runs as various `URLSession` requests are made and received. When the last image is added to `images` the `Task` completes.

Before we use `TaskGroup` which is the correct tool for our job, let's look at what happens if we try to use `async let`.

Structured Concurrency with async let

Earlier in the chapter we used `async let` to create child `Task`s.

Let's try this technique when we fetch the images.

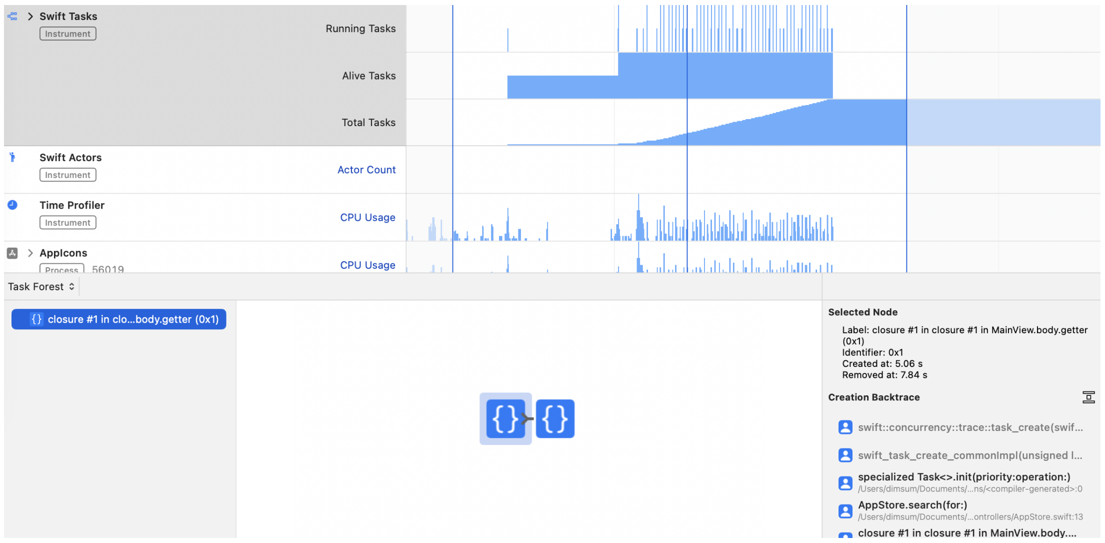
*AppIcons/Controllers/**AppStore.swift***

```
extension AppStore {
  private func retrieveImages() async throws {
    for app in apps {
      async let (imageData, _)
      = ~~try await~~ ephemeralURLSession
        .data(from: app.artworkURL)
      let image = UIImage(data: try await imageData)
      publish(image: image,
              forAppNamed: app.name)
    }
  }
}
```

Run the app and you'll see the icons appear in the same order filling rows from top to bottom with each row being filled from left to right.

The `for` loop takes each `app` in turn and creates a child `Task`. The next `Task` isn't created until `imageData` returns and we `publish()` the image.

Run the app using the Swift Concurrency instrument. You should see something like this.



```
Selected Node
    Label: closure #1 in closure #1 in MainView.body.getter
    (0x1)
    Identifier: 0x1
    Created at: 5.06 s
    Removed at: 7.84 s
Creation Backtrace
    swift::concurrency::trace::task_create(swif...
    swift_task_create_commonImpl(unsigned l...
    specialized Task<>.init(priority:operation:)
    /Users/dimsum/Documents/...ns/<compiler-generated>:0
    AppStore.search(for:)
    /Users/dimsum/Documents/...ontrollers/AppStore.swift:13
    closure #1 in closure #1 in MainView.body....
```

You can see a single `Task` when the initial request is made. After it returns we start retrieving images using child `Task`s. Notice that only a single child `Task` is alive at any time. When the final image is retrieved we drop from two `Task`s to none.

You can also see in the Task Forest that there is only one child `Task` at a time.

There is a tool that allows us to do more work at once and that is the `TaskGroup`.

Introducing TaskGroup

We create a `TaskGroup` with a helper function in much the same way that we created a `CheckedContinuation`. The trailing closure in `withTaskGroup()` or `withThrowingTaskGroup()` provides the group that we'll be using.

`withThrowingTaskGroup()` is `async throws` because the method waits for all of the `Task`s in the `group` to complete or for one to throw an error.

When we create a `TaskGroup` we have to specify what it's a `TaskGroup` of.

Unlike `AsyncStream`, we aren't specifying what will be emitted by the `TaskGroup`. A `TaskGroup`, as you may expect, contains `Task`s.

But each child `Task` may return a value. We'll see how that works in the next section. For now the `Task`s don't return anything which is why we've specified that we're creating throwing `TaskGroup`s of `Void.self`.

*AppIcons/Controllers/**AppStore.swift***

```
extension AppStore {
  private func retrieveImages() async throws {
    try await withThrowingTaskGroup(of: Void.self) { group in
      for app in apps {
        async let (imageData, _)
        = try await ephemeralURLSession
          .data(from: app.artworkURL)
        let image = UIImage(data: try await imageData)
        publish(image: image,
                forAppNamed: app.name)
      }
    }
  }
}
```

We have a `TaskGroup` but it isn't yet doing anything. Run the app and it runs as before.

Adding Tasks to a TaskGroup

Add child `Task`s to our `group` with `group.addTask()`.

The work that each `Task` will perform is specified in the trailing closure. In our current case each iteration of the `for` loop is it's own child `Task`.

The order in which `Task`s in a group will be run is explicitly not guaranteed.

There are two compiler errors with the following that we'll fix in a minute.

*AppIcons/Controllers/**AppStore.swift***

```
extension AppStore {
  private func retrieveImages() async throws {
    try await withThrowingTaskGroup(of: Void.self) { group in
      for app in apps {
        group.addTask {
          async let (imageData, _)
          = try await ephemeralURLSession
            .data(from: app.artworkURL)
          let image = UIImage(data: try await imageData)
          publish(image: image,   // problem
                  forAppNamed: app.name)
        }
      }
    }
  }
}
```

We're calling `publish()` from inside a child `Task`. It doesn't get the context of `self` that the parent `Task` has so we have to use `self.publish()`.

In addition, `publish()` is on the main actor but the call to `publish()` is not on the main actor so we have to use `await` with `publish()` because the main actor might be busy doing something else.

We also have a warning that no calls to throwing functions occur within the `try` so we can eliminate it.

AppIcons/Controllers/***AppStore.swift***

```
extension AppStore {
  private func retrieveImages() async throws {
    ~~try~~ await withThrowingTaskGroup(of: Void.self) { group in
      for app in apps {
        group.addTask {
          async let (imageData, _)
          = try await ephemeralURLSession
            .data(from: app.artworkURL)
          let image = UIImage(data: try await imageData)
          await self.publish(image: image,
                             forAppNamed: app.name)
        }
      }
    }
  }
}
```

Run the app and enter a search term.

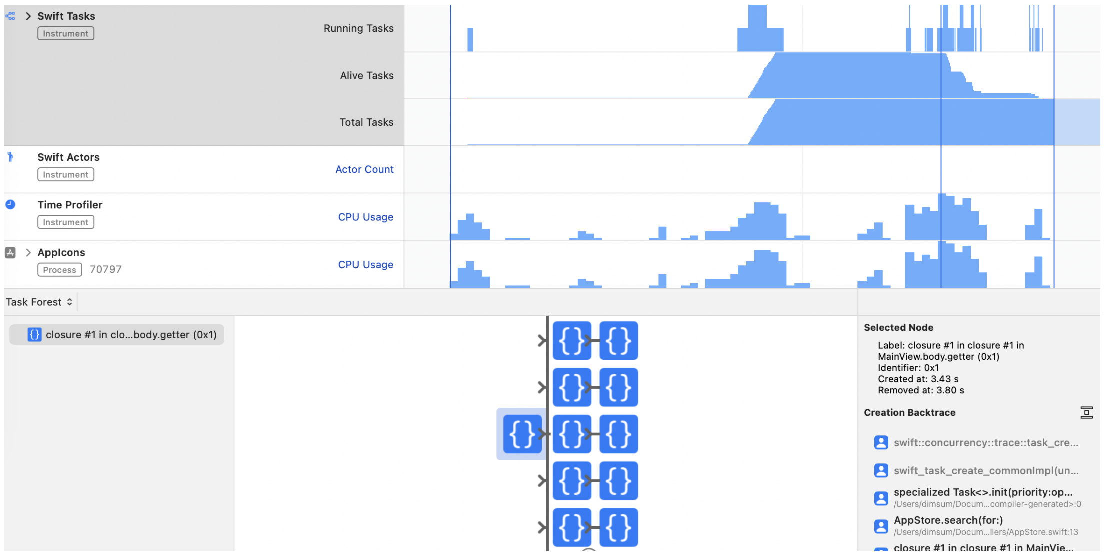
You should notice two things. First, the icons appear much more quickly. Second, the icons no longer appear in an orderly fashion.

The "After" picture

Run the app in the Swift Concurrency instrument again and enter a search term.

You can see empirically how much faster the images return.

Once again look at the alive and running `Task`s and look at the Task Forest.



Selected Node

Label: closure #1 in closure #1 in MainView.body.getter (0x1)
Identifier: 0x1
Created at: 3.43 s
Removed at: 3.80 s

Creation Backtrace

swift::concurrency::trace::task_cre...
swift_task_create_commonImpl(un...
specialized Task<>.init(priority:op...
/Users/dimsum/Docum...compiler-generated>:0
AppStore.search(for:)
/Users/dimsum/Docum...llers/AppStore.swift:13
closure #1 in closure #1 in MainVie...

The TaskForest shows that each of the `Task`s added to the `TaskGroup` is a child `Task`. We can also see that each of these has a further child `Task` because we use `async let`. If you change this back to `try await` instead of `async let` we'll only have one level of child `Task`s.

Second, look at the graphs for running and alive `Task`s. You can see the initial `Task` created. Once the `data()` returns and we create group `Task`s for each image data request, the number of alive `Task`s grows. Then there's another pause while we wait for these to return. Following the pause, the number of running `Task`s spikes and the number of alive `Task`s starts to fall as they complete.

In the next section we take advantage of something very cool about `TaskGroup`s: they are `AsyncSequence`s.