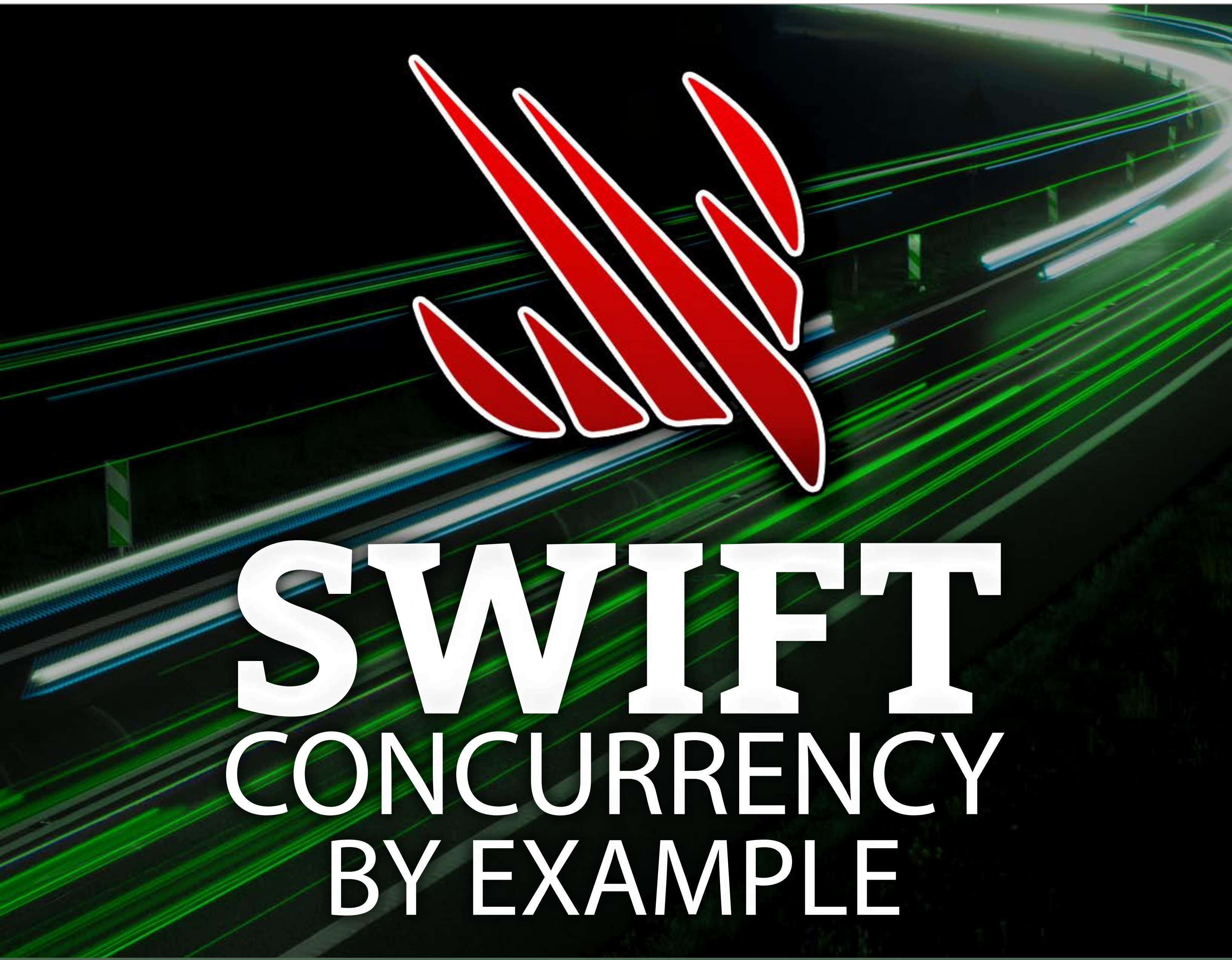
HACKING WITH SWIFT



SWIFT CONCURRENCY BY EXAMPLE

COMPLETE REFERENCE GUIDE

What it does, how it works
best practices for your ects

FREE SAMPLE

Paul Hudson

Chapter 1
Introduction

This stuff is hard

You might think this is a strange way to start a book, but: concurrency is a hard topic, and you might sometimes find yourself feeling a bit lost or a bit confused while you're learning about it.

I'm not saying that to put you off, or to make you feel somehow scared by all the topics we're going to cover. In fact, my goal here is quite the opposite: I want you to know that *everyone* finds this stuff hard. If you ever feel like you're struggling, it's okay – it's not you being stupid, it's just plain difficult stuff.

It's my goal to try to make this all as straightforward, understandable, and most importantly *applicable* as I can, so that anyone who is an intermediate to advanced Swift developer can apply these concepts to their projects immediately. If you're a beginner developer you're welcome to try following along, but honestly I would suggest you come back later – lots of theory isn't likely to stay in your head unless you have the chance to actually apply it.

Our brains are inherently not built to work concurrently, and as a result it's often hard to think about. So, one last time: this stuff is hard, and if you're finding it hard it's just a sign your brain is working correctly.

How to follow this guide

This guide is called Swift Concurrency by Example because it focuses on providing as many examples as possible. My goal is to stay laser-focused on real-world problems and real-world solutions, and to give clear, pragmatic advice on how to use specific techniques and approaches.

A lot of the time I've tried to make the chapters in this book answer specific questions such as "what are…" or "how to…", so that you can see exactly what problem is being solved and get straight there. That also means I've tried to get to the point as fast as possible and stay there so that you can get answers and build understanding quickly.

You can read this in a linear order if you want, or just dive in to a particular chapter that interests you – either one works.

Concurrency vs parallelism

When working through concurrency in Swift, there are two words we use a lot: *concurrency* and *parallelism*. We give them very specific meanings that might not quite match how you use them in English, so it's worth clarifying up front what they mean in this specific context.

Imagine a single-core CPU running a desktop operating system: the user can run lots of programs, use the internet, maybe play some music, and from the user's perspective all those things are happening at the same time. However, we know that isn't possible: they have a single-core CPU, which means it can literally only do *one* thing at a time.

What you're seeing here is called *concurrency*: our user's apps have been written in such a way that one CPU core can juggle them so they appear to be running at the same time. One starts, makes a tiny bit of progress on its work, then pauses so that another one can start, make a bit of progress on *its* work, then pause, and so on. Because the CPU flips between programs so quickly, they appear to be running all at the same time, when really they aren't.

As soon as you add a *second* CPU core to a computer, then things can run in *parallel*. This is when two or more programs run at the same instant: one on the first core, and another on the second. As you might imagine, this means work can happen up to twice as fast because two programs are able to run at the same time.

The really interesting stuff lies in our ability to split up work in a single program. Yes, having two CPU cores allows a computer to run two separate programs at the same time, but we can also split up our work into smaller parts called threads, and *those* can be run in parallel too.

This splitting up of functionality requires us to do work – Swift can't decide by itself to run some parts of our code in parallel with other parts, because that would introduce a lot of surprising bugs. Instead, we have to tell Swift ahead of time which parts of our code can be split up if needed, and also tell it what we should do when those tasks complete.

When you boil it right down, that's the topic of this whole book: teaching Swift how it can split up the work in our programs so it runs as efficiently as possible. And it *is* about efficiency, because some Apple devices have many CPU cores – if your app is running full screen on that device and you’re only ever using one of those cores, you’re only getting a tiny fraction of the device’s possible performance.

More importantly, you’re also helping to make sure your app remains responsive the entire time. Imagine if your user interface froze up every time you were waiting for the response to a network request – it would be a pretty horrible experience, right?

There’s a famous computer scientist called Rob Pike, and I think he explained the difference between concurrency and parallelism beautifully. Here’s what he said:

“Concurrency is about dealing with many things at once, parallelism is about doing many things at once. Concurrency is a way to structure things so you can maybe use parallelism to do a better job.”

Understanding threads and queues

Every program launches with at least one thread where its work takes place, called the *main thread*. Super simple command-line apps for macOS might only ever have that one thread, iOS apps will have many more to do all sorts of other jobs, but either way that initial thread – the one the app is first launched with – always exists for the lifetime of the app, and it's always called the main thread.

This is important, because all your user interface work must take place on that main thread. Not some work some of the time, but *all* work *all* the time – if you try to update your UI from any other thread in your program you might find nothing happens, you might find your app crashes, or pretty much anywhere in between.

This rule exists for all apps that run on iOS, macOS, tvOS, and watchOS, and even though it's simple you will – *will* – forget it at some point in the future. It's like an initiation rite, except it happens more often than I'd like to admit even after years of programming.

Although Swift lets us create threads whenever we want, this is uncommon because it creates a lot of complexity. Each thread you create needs to run *somewhere*, and if you accidentally end up creating 40 threads when you have only 4 CPU cores, the system will need to spend a lot of time just swapping them.

Swapping threads is known as a *context switch*, and it has a performance cost: the system must stash away all the data the thread was using and remember how far it had progressed in its work, before giving another thread the chance to run. When this happens a lot – when you create many more threads compared to the number of available CPU cores – the cost of context switching grows high, and so it has a suitably disastrous-sounding name: *thread explosion*.

And so, apart from that main thread of work that starts our whole program and manages the user interface, we normally prefer to think of our work in terms of *queues*.

Queues work like they do in real life, where you might line up to buy something at a grocery store: you join the queue at the back, then move forward again and again until you're at the front, at which point you can check out. Some bigger stores might have lots of queues leading up to lots of checkouts, and small stores might just have one queue with one checkout. You might occasionally see stores trying to avoid the problem of one queue moving faster than another by having a single shared queue feed into multiple checkouts – there are all sorts of possible combinations.

Swift's queues work exactly the same way: we create a queue and add work to it, and the system will remove and execute work from there in the order it was added. Sometimes the queues are *serial*, which means they remove one piece of work from the front of the queue and complete it before going onto the next piece of work; and sometimes they are *concurrent*, which means they remove and execute multiple pieces of work at a time. Either way work will start in the order it was added to the queue unless we specifically say something has a high or low priority.

You might look at that and wonder why you even need serial queues – surely running one thing at a time is what we're trying to avoid? Well, no. In fact, there are lots of times when having the predictability of a serial queue is important.

As a simple example, your user might want to batch convert a collection of videos from one format to another. Video conversion is a really intense operation and is already highly optimized to take full advantage of multi-core CPUs, so it's more efficient to convert one video fully, then the next, then a third, and so on until you reach the end, rather than trying to convert four at once. This kind of work is perfect for a serial queue.

More importantly, sometimes serial queues are *required* to ensure our data is safe. For example, manipulating your user's data is exactly the kind of work you'd want to do on a serial queue, because it stops you from trying to read the data at the same time some other part of your program is trying to write new data.

Putting this all together, threads are the individual slices of a program that do pieces of work, whereas queues are like pipelines of execution where we can request that work be done at some point. Queues are easier to think about than threads because they focus on what matters:

we don't care how some code runs on the CPU, as long as it either runs in a particular order (serially) or not (concurrently). A lot of the time we don't even create the queue – we just use one of the built-in queues and let the system figure out how it should happen.

Tip: Sometimes Apple's frameworks will help you a little here. For example, even though using the **@State** property wrapper in a view will cause the body to be refreshed when the property is changed, this property wrapper is designed to be safe to call on any thread.

Main thread and main queue: what’s the difference?

The main thread is the one that starts our program, and it’s also the one where all our UI work must happen. However, there is also a main *queue*, and although sometimes we use the terms “main thread” and “main queue” interchangeably, they aren’t quite the same thing.

It’s a subtle distinction, but it can sometimes matter: although your main queue will always execute on the main thread (and is therefore where you’ll be doing your UI work!), it’s also possible that other queues might sometimes run on the main thread – the system is free to move things around in whatever way is most efficient.

So, if you’re on the main queue then you’re definitely on the main thread, but being on the main thread doesn’t automatically mean you’re on the main queue – a different queue could temporarily be running on the main thread.

At this point you’re very likely staring at the screen wondering when this would ever be a problem, or perhaps even rereading what I said like it’s a cryptic riddle. Trust me, if you ever hit a problem where the main thread vs main queue matters, you’ll be glad you read this!

Where is Swift concurrency supported?

When it was originally announced, Swift concurrency required at least iOS 15, macOS 12, watchOS 8, tvOS 15, or on other platforms at least Swift 5.5.

However, if you're building your code using Xcode 13.2 or later you can back deploy to older versions of each of those operating systems: iOS 13, macOS 10.15, watchOS 6, and tvOS 13 are all supported. This offers the full range of Swift functionality, including actors, async/await, the task APIs, and more.

Important: This backwards compatibility applies only to Swift language features, not to any APIs built using those language features. This means you can write your own code to use async/await, actors, and so on, but you won't automatically gain access to the new Foundation APIs using those – things like the new **URLSession** APIs that use async/await still require iOS 15.

If you are keen to use the newer APIs in your project while also preserving backwards compatibility for older OS releases, your best bet is to add a runtime version check for iOS 15 then wrap the older APIs with continuations. This kind of hybrid solution allows you to keep using async/await elsewhere in your project – you get all the benefits of concurrency for the vast majority of your code, while keeping your backwards deployment shims neatly organized in one place so they can be removed in a year or two.

Dedication

This book is dedicated to the many people who helped read through earlier editions, reporting typos, suggesting improvements, and helping make this book as comprehensive as it can be – I'm really grateful they took the time to provide so much help and support!

In alphabetical order, they are: Abdul Ajetunmobi, Alejandro Martinez, Andrew Cowley, Chad Naeger, Chris Rivers, Darius Dunlap, Devanshi Modha, Diego Freniche Brito, Eric Schofield, Glenn Sequeira, Gustavo Diel, Hamish Allan, Horacio Chavez, Howard Katz, Isa Hashim, Jana Grill, Jason Bruder, Jeff Terry, Lars Christiansen, Laurent Brusa, Tor Rafsol Løseth, Mats Braa, Michael M. Mayer, Paul Williamson, Phil Martin, Piers Ebdon, Rick Gigger, Sarah Reichelt, Thomas Alvarez, Tom Comer, Vikash Anand, and Zack Apiratitham.