Surviving RxSwift

Best practices and how to get your head around reactive programming in Swift

Ian Keen
May 24, 2019 · 6 min read ★

Photo by Sarah Pflug from Burst

Before I start I need to mention that these are a set of best practices that I *personally* use to get the most out of RxSwift (Rx) and avoid many of the common pitfalls. They are by no means a set of hard rules that will work for everyone, but I encourage you to try them.

Why use Rx?

If so many people struggle with Rx, why use it at all? This is a perfectly valid question.

In our profession there are many ways to solve a problem and Rx is no different. Rx has a steep learning curve and it often isn't clear if one way of solving something is better than another. Figuring out the subtleties will take time, but it is very rewarding when you do.

The reason I use Rx lies in its greatest strength: the ability to take inputs from many sources, tame the often complex logic needed to combine them, and leave you with outputs that update whenever the inputs do.

What this means is, if you are following principles like SRP, you often don't need Rx for the individual, lower level components. However, its power becomes quite clear in higher level ones.

With that in mind, let's get started.

Use it sparingly

The most problematic Rx codebases I have worked on and heard people complain about are the ones that have made, or tried to make, *every single part* reactive. The more Rx you use the bigger and scarier your stack traces are. Stepping through a long reactive chain is *hard* and the `.debug()` operator only gets you so far.

As I alluded to earlier, my solution to this is *only* using Rx in the higher level components such as view models and view controllers.

But…how do we make all these non-Rx things work in Rx?

Wrap components in Rx

We can build components, such as an API layer, using more widely understood paradigms like closures to deliver things asynchronously. For example:

To make this work nicely with Rx we only need a few lines of code:

With this, our `APIClient` can now easily be used in Rx chains. So what is the benefit of doing it this way?

Debugging is now much easier, we can set breakpoints within our non-Rx code and step through without all the additional noise Rx adds to our stack traces. We can also write tests using the standard XCTest frameworks. Finally, we have a much more portable component — there is now nothing stopping us from using this same API layer in other apps that may not use Rx.

You don't need most traits

RxSwift provides a wide range of traits. Traits are wrappers around a standard `Observable` that provide additional semantics/behaviours. This might sound great — we want to leverage Swifts' type system after all right? It's quite common to see code using a `Single` rather than an `Observable` for the API layer example above.

The problem with these different traits is that they do not always compose well together. There are a number of custom extensions floating around to help translate between them. You can avoid jumping through all these hoops by simply sticking with a standard `Observable`. There is no downside to avoiding traits.

Keen-eyed readers will have spotted the "most" in the heading. There is an exception to this one! You 100% should be using RxCocoa traits like `Driver` when you expose values for your view to consume as well as `ControlEvent` or `ControlProperty` which you encounter when using properties like `UIButton.rx.tap`. This is because those traits are designed specifically to interact with your UI. They will ensure values are shared and that everything happens on the main thread.

Avoid Subjects

There are a number of shifts you need to make in how you think about data when learning Rx. One of the biggest is learning to construct streams such that the values coming *out* of them are the result of combining and/or transforming the values being sent *in*. A simplistic way to think about streams is like a mapping function `(A) -> B`, or `(A, B) -> C`

Quite often when that shift has not yet been made you will see code that falls back on subjects to bridge the gap. Subjects are objects in Rx that are both input *and* output. They can be subscribed to but also incoming values can be sent to them.

Let's look at one such example of a view model for performing a search:

On its own, this looks quite straightforward. `searchText` receives some text and we use that to do our search. From that, we can define two other streams, one for our list of results and one for a string showing the count.

However, because `searchText` is both input *and* output we have accidentally introduced an additional output to our view model. There is no way for us to prevent something else from subscribing to `searchText` and executing other code. If this is something you need to do then you should either:

- Subscribe directly to the underlying input source (i.e. the text field); or
- Create another explicit output for this behaviour

By not creating a clear separation between input and output it becomes harder to debug where values may be coming from. It's also harder to reason about what the inputs and outputs for your system actually are.

Now let's see how we might refactor the same view model to remove the use of subjects:

The changes here are incredibly subtle. We have required that the input is passed via the constructor. This change gives more fine-grained control over what the inputs and outputs are. We now have a single starting point to search for bugs relating to both text coming in and search results going out.

> It's worth noting that there are, of course, other ways you can go about removing/encapsulating the subjects.

Again, there are exceptions to this one too but I'm not going to go into that here… If you absolutely feel that you need to use subjects, at the very least ensure you are encapsulating them so they are not publicly exposed for any random code to interact with. I would, however, encourage you to try and avoid them if you can!

Limit DisposeBags to View Controllers

Another issue you will see before the shift I mentioned has been made is `DisposeBag` s in places they don't belong. A `DisposeBag` is something that holds subscriptions. Subscriptions are what you get when you use functions like `subscribe` , `drive` or `bind` . These subscriptions are also where your apps side-effects are contained.

These subscriptions should be limited to your view controllers. Other components like view models should not contain subscriptions because they should not contain side-effects.

Interestingly enough, this problem is quite often seen when subjects are also being misused, as mentioned above. It is common to see subscriptions calling functions like `onNext` directly. This is a sign that your stream outputs are not being modelled as transformations of their inputs.

As with the others, there are rare exceptions to this one too. If, for some reason, a stream does not have an output you may need a single private subscription to ensure the stream is enabled. However, as with the other exceptions, do your best to avoid this situation.

Summary

So to quickly sum up:

- Limit Rx to higher level components like view models and view controllers.
- Build lower level components without Rx then add Rx wrapper extensions.
- Limit yourself to `Observable` and `Driver` .
- Write your Rx without subjects as much as possible.
- `DisposeBag` s really only belong in view controllers.

I would strongly suggest to anyone new to Rx or is feeling overwhelmed by it to simplify your use with these tips. As you get more comfortable with Rx you will start to get a sense of when and how these tips apply, as well as when they don't.

Thanks to Shai Mishali.

Swift Programming Software Development Software Engineering iOS