# Stateless Cross-Platform Health Data Access

It was a case of absolute serendipity. A beautiful solution for a major cross-platform architectural problem wafted like a feather from the sky and landed softly on the top of my head.

I was working on a way to do shared access of health data from HealthKit on iOS, and fitness data from the Google Fit API, via Xamarin, in a shared and cross-platform manner.

I wanted a set of shared helper objects between iOS and Android that would relay the health state (i.e. weight, height, glucose level, steps taken) to respective UIViewController and Activity objects.

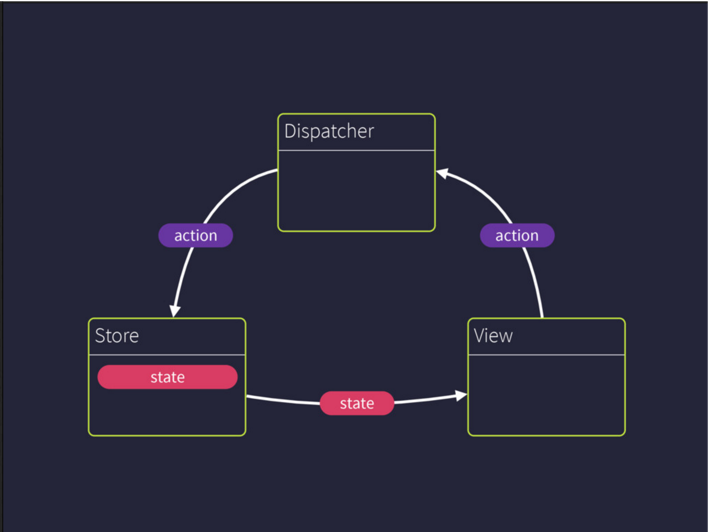
I had an idea, and messy code implementation, that would read up the data and populate objects with the retrieved health data from their respective platforms. But the whole structure was all wrong:

* Not enough code was being shared between iOS and Android.
* Too much of the HealthKit stuff was leaking through to the UIViewControllers.
* Too much of the Google Fit stuff was leaking through to the Activity classes.
* UIViewControllers / Activities were mutating state directly and independently.
* My Table Views on iOS were an absolute mess!
  + Tons of repeated code
  + All the classic mess of ‘section counts’ and ‘row count’ retrieval.
  + Raw HealthKit stuff leaking through on row insert and row delete.
  + [Dogs and cats living together! Mass hysteria!](https://www.youtube.com/watch?v=WfVcvyxLj-s)
* The ListAdapters on Android were just as much of a mess.
* The 2 list types between iOS and Android were different and goofy.
* There was no clean way to get lists of data (i.e. step counts) in places other than UITableViewControllers / ListAdapters.

It was a total yuck fest. That was Friday 4/22/2016. Down trodden, and with fewer hairs on my head at 5PM than at 6AM, I headed home.

Then Saturday 4/23/2016 came around. On 4/23/2016 the [minnebar 11 conference](http://minnestar.org/minnebar/) was held at Best Buy Headquarters. I was lucky enough to attend a session titled ‘[Exploring Stateless UIs in Swift](http://sessions.minnestar.org/sessions/350)’ being held by Adam May and Sam Kirchmeier. Being in their session was like watching `The Cat In The Hat` from Dr. Seuss take the whole mess of that bad Friday code and organize it perfectly before mom got home.

It all appeared from one diagram, in a single blinding flash, during Adam and Sam’s presentation:



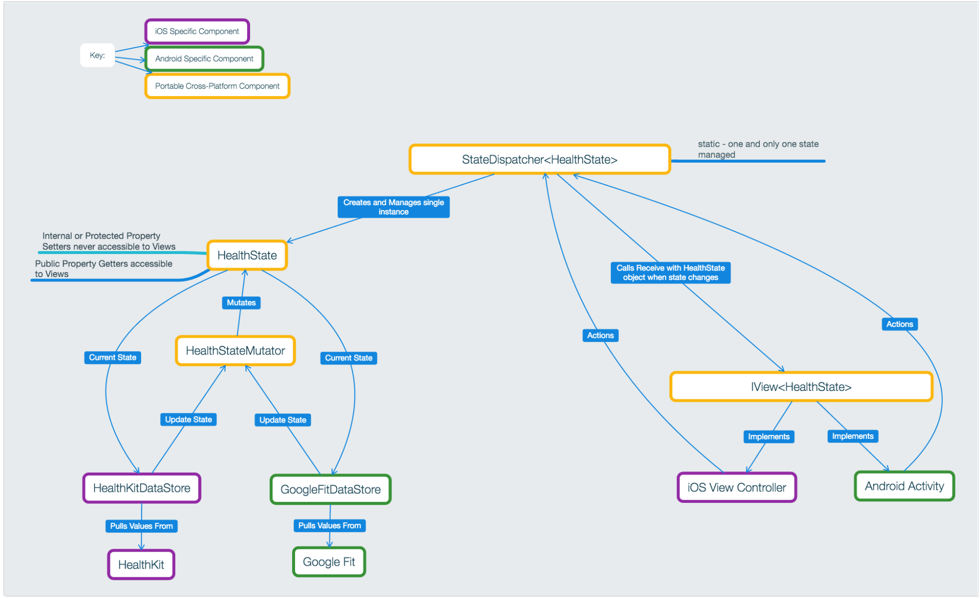
I can’t really do justice to Adam and Sam’s [presentation](https://speakerdeck.com/skirchmeier/exploring-stateless-uis-in-swift) in a non-20 page document. But let me see if I can summarize the key design pieces that I latched onto to solve my issues:

* iOS View Controllers (and by extension Android Activity classes) become a View.
  + A view never directly mutates data.
  + A view sends abstract user gestures to a Dispatcher.
  + A view receives broadcasted data via a single method call. A parameter to that method call is a State object sourced from a Data Source object with no ability to mutate the original data.
* A Dispatcher is responsible for processing all gestures from a View and relaying these gestures to 1 or more Data Source objects.
* A Data Source object is responsible for mutating state based on gestures, and broadcasting the new State objects to the Views.
* It helps if the Dispatcher and Data Store objects are purely static.
* To help with Views that show lists of data (i.e. UITableViewController on iOS) a Data Source would broadcast out a changeset that indicates the exact changes in the list since the last update.
  + Note: The list / changeset scheme is not covered in this post, but I did write fully functioning C# code that creates and processes these changesets. Feel free to take a look.

Adam and Sam’s presentation was able to use the core paradigms in the Swift language to enforce their beautiful stateless structure. However, I had this pile of code all in C# against Xamarin.iOS / HealthKit access via Xamarin.iOS shims, Mono.Android / Xamarin shims for Google Fit access. I didn’t have the same advantages of Swift language constructs, but I did have plenty of cool tools I could use from C#.

## Starting with the end in mind

I am going to show you my [end product first](https://github.com/Batgar/HealthKitStatelessSample/blob/master/Stateless Cross-Platform Health Data Infrastructure.pdf) (PDF at link):



[Info overload! Run away!](https://www.youtube.com/watch?v=7FPELc1wEvk)

What happened to that clean diagram that Adam and Sam had? Well, it got ran through the ‘Dan Filter’ and turned into something insane.

For even more info overload, take a look at my [repo out on Github](https://github.com/Batgar/HealthKitStatelessSample).

## My State Dispatcher Hack!

With all apologies to Adam and Sam, I took a series of massive shortcuts to arrive at a working implementation.

The largest shortcut was the “State Dispatcher”. You may notice that there is no connection between the Data Store objects and the Views. My massive shortcut was to make a Dispatcher a “State Dispatcher”.

In my C# oriented static generic world I wanted to be able to seamlessly bootstrap the state objects by type. I chose the Dispatcher from the original design to be the bootstrapper for the State objects.

To make this happen I transmogrified the Dispatcher into a StateDispatcher<T>, where T is a type of State object that would be created by the StateDispatcher<T>, passed to a View for updates, and mutated by a Data Source.

This elevates the Dispatcher to be the static source of a single State object instance as well as the broker of View gestures and View state changes.

It also elevates the State object type to a core identifier across the entire architecture. Another way to think of it is that State object type becomes a kind of ‘line protocol’ that everything in the system uses to communicate and bind together (including the compiler).

For example, I have a HealthState object:

public class HealthState : IState {

public double Height { get; internal set;}

public double BloodGlucose {get; internal set;}

…

}

The HealthState object is now the ‘coin of the realm’ throughout the kingdom ruled by StateDispatcher<HealthState>.

The HealthState object is the ‘protocol’ that will be used to broadcast out **Height** and the last **BloodGlucose** value to all HealthState dependent views.

I also want specific views to be bound to just this HealthState object to make view code cleaner.

The HealthState object is passed onto Activity classes and UIViewControllers by having those classes implement IView<HealthState>:

partial class MainActivity : **IView<HealthState>**

{

//Implementation of IView<HealthState> to show the HealthState members in the Android Activity.

**public void Receive(HealthState healthState)**

{

var height = healthState.Height;

var textView = FindViewById<TextView> (Resource.Id.heightInMeters);

textView.Text = string.Format("Height: {0}", height);

}

}

The HealthState object will get the Height and BloodGlucose data from either HealthKit (on iOS) or Google Fit (on Android) via a platform specific Data Source for each.

In order for the per-platform Data Source objects to modify the HealthState, they go to StateDispatcher<HealthState> to get the current HealthState object and then apply a special mutator object to it:

//Mutate the height property in the singleton HealthState object.

HealthStateMutator.MutateHeight(**StateDispatcher<HealthState>.**State,

() => heightQuantityFromHealthKit.GetDoubleValue(heightUnit));

//Signal all views that the singleton HealthState object has new data to show.

StateDispatcher<HealthState>.Refresh();

## State Object Mutation – Yet another shortcut I took

Let’s talk about mutation of the HealthState object:

public class HealthState : IState {

public double Height { get; **internal** set;}

public double BloodGlucose {get; **internal** set;}

…

}

You may be wondering: Why the ‘internal’ keyword on the property setters?

In this design wanted to use 1 and only 1 instance of the State type to go across Data Source, Dispatcher, and Views.

However, the last thing I ever wanted was for the View and Dispatcher to change the state of this single instance State object while it is in flight.

I ensure that any mutation on the State class is done via a special mutator object that has to be revved up and sourced ONLY from the same assembly as that which hosts the HealthState object (hence internal property access).

In some other cases (i.e. BloodGlucoseEntry) the property sets are marked protected so that they can only be modified from an inherited form of the given State object.

I left it up to the author of the state object to provide direct mutation functions off of the class. That said: I warn you in advance. If you put public mutating functions on a class, then someone WILL call them from places you never want them called from! Don’t expose mutation to any part of the hierarchy! EVER! Trust no one with mutation! This is a source of bugs, misery, lateness, and mad managers!

In all honesty, I took some shortcuts on mutation in the name of readability and as a compromise to have a StateDispatcher<T> as the bootstrapper for everything in the architecture. We can probably do better, but this mutation scheme seems OK for now. There are enough guards to stop mutation from occurring in places where it shouldn’t (even accidentally).

Remember: [Shipping is a feature](http://a16z.com/2014/04/16/shipping-is-a-feature-some-guiding-principals-for-people-that-build-things/).

## Any State Object Will Do!

Another crazy side effect of creating a StateDispatcher<T> is that we can transplant any object as a State object provided that it meets the restriction that it inherits from IState.

Note: If you look at [IState](https://github.com/Batgar/HealthKitStatelessSample/blob/master/Stateless/IState.cs) you will scratch your head and curse my name. IState is just a type placeholder that is meant to keep you honest and on track regarding object typing. In the future IState could take on more meaning… **Evil Laugh**

Let’s show all of the above HealthState code, only now with a totally different object type as the state object.

We will pivot away from HealthState and use a BloodGlucoseRecommendation object for state. This new object takes in the glucose level and will provide a user displayable string based on the value:

if (\_bloodGlucose < 80) {

RecommendationText = "Dude, you need to get some candy! Stat!";

} else if (\_bloodGlucose < 120) {

RecommendationText = "All is cool";

} else {

RecommendationText = "Whoa! You have really high blood sugar. Might be time for insulin!";

}

Note: If you are taking any of the above as true diagnostic medical advice, then you need to seek out the advice of a psychologist as well as a physician.

To reiterate, the type of State object put into the architecture dictates the ‘line protocol’ that is used across the whole architecture.

## Partial Classes for your Views – Activity and UIViewController objects

One great side effect of using an IView<T>, where T is the type of any state object, is that you can create separate source files for each of your views which contain a partial class that implements your IView<T>.

Android Activity – Partial class:

**partial** class MainActivity : **IView<BloodGlucoseRecommendationState>**

{

public void Receive(BloodGlucoseRecommendationState state)

{

var recommendationText = state.RecommendationText;

//TODO: Send recommendation text to a label on the view.

}

}

iOS UIViewController – Partial class:

**partial** class ViewController : **IView<BloodGlucoseRecommendationState>**

{

public void Receive(BloodGlucoseRecommendationState state)

{

this.bloodGlucoseRecommendation.Text = state.RecommendationText;

}

}

Having partial classes per IView<T> really frees up your code, and serves as a nice isolation and discovery point for how your UI is responding to state objects of different types.

The partial class is also one more tool to help mitigate the problem of the MVC (Massive View Controller) on iOS.

It also really helps solidify exactly what changed within the system when you take a look at your Git commits.

## Binding your IView<T> to the StateDispatcher<T>

How does the StateDispatcher<T> know to call your slick IView<T>.Receive implementation?

All you need to do is go to the static StateDispatcher<T> and call Bind with your IView<T> implementation:

partial class ViewController

{

private void Bind()

{

StateDispatcher<HealthState>.Bind(this);

StateDispatcher<BloodGlucoseRecommendationState>.Bind (this);

//Future StateDispatcher bindings go here!

}

private void Unbind()

{

StateDispatcher<HealthState>.Unbind(this);

StateDispatcher<BloodGlucoseRecommendationState>.Unbind (this);

//Future StateDispatcher un-bindings go here!

}

}

In the case above, I created a partial class and threw my bindings into private methods.

Did you also see that a single class can be a view of many different types of state objects? So cool!

Those private methods are then called in a good place within the lifetime of your Activity or UIViewController:

public partial class ViewController : UIViewController

{

//SNIP!

public override void ViewDidLoad ()

{

base.ViewDidLoad ();

// Perform any additional setup after loading the view, typically from a nib.

**Bind ();**

}

public override void ViewDidUnload ()

{

base.ViewDidUnload ();

**Unbind ();**

}

//SNIP!

}

In my implementation of StateDispatcher<T>.Bind, I put in code that will immediately call back onto the IView<T>.Receive method when it is bound. This immediate call back to Receive on StateDispatcher<T>.Bind ensures that the view will be automatically refreshed with whatever the current values in the given single state object are.

Future enhancements to the Bind / Unbind in StateDispatcher<T> will that the storage of your instance is weak so our evil top level static isn’t leaking instances of your IView<T> implementations.

## Anything can be an IView<T>

Just to note: There is absolutely nothing stopping you from having any class inherit from IView<T> and bind to your StateDispatcher<T>.

You do not have to constrain the IView<T> inheritance + StateDispatcher<T> Bind calls to just Activity and UIViewController objects.

For example: You could have a class called MasterLogger that is an IView<… Every State …> and Binds to the matching StateDispatcher<… Every State ...> and logs every broadcasted state transition within your app.

The cross-cutting concerns regarding state that you can isolate, contain, and centrally manage are only limited by your imagination.

## Async Data Sources

There is nothing in this design that stops you from creating long running status objects that serve up state as it arrives asynchronously.

For example: You can have a NetworkState object and a NetworkDataSource object that your app revs up on startup. Then views could be IView<NetworkState> by binding to the StateDispatcher<NetworkState>.

The one caveat is that you should ensure that you marshal any async callbacks back to your UI thread. On Xamarin, this can be done via SynchronizationContext, Activity.RunOnUiThread (Android), or DispatchQueue.MainQueue.DispatchAsync (iOS).

In my case, since my Data Source objects are abstract per-platform and bootstrapped per-platform I just ensure that any StateDispatcher<T>.Refresh() calls happen within a closure for one of the above synchronizing calls.

## Side note: Lists of objects and UITableViewController

A major part of Adam and Sam’s presentation related to simplifying the UITableViewController implementation.

I took an initial stab at trying to do something similar with this implementation.

It is still largely under construction. For the brave and the bold, feel free to take a look at the code. It works! I promise. But it is kinda complicated. We will talk about Lists + UITableViewController (iOS) + ListActivity (Android) + ListAdapter (Android) simplification at some other time.

Feel free to start looking at [StepCountTableViewController.View.cs](https://github.com/Batgar/HealthKitStatelessSample/blob/master/HealthKitSample/StepCountTableViewController.View.cs), watch as it tracks through [GeneralListViewTableViewController.cs](https://github.com/Batgar/HealthKitStatelessSample/blob/master/HealthKitSample/GeneralListViewTableViewController.cs), and onto [ListStateDispatcher.cs](https://github.com/Batgar/HealthKitStatelessSample/blob/master/Stateless/ListStateDispatcher.cs)

Go ahead. Put breakpoints in. Add some step counts + blood glucose readings. It will work.

The above source file links may be broken depending on how much refactoring I get done on that preliminary solution before the next post.

## Futures

This code sample is just a sample of the power of the role that a stateless pattern can take in your mobile user interface code.

Much more cleanup, simplification, and generalization is possible.

We haven’t even begun to scratch the surface of how this design could be refactored, wrapped, and stretched out even more.

My hope is that you enjoyed the concepts, and the sample code, and that this can serve as future cross-platform stateless inspiration for your next project.