Given the paucity of cryptocurrency data available in an easily accessible format, it was quite difficult to say for certain that a particular investment was a good one relative to some alternative, unless you were very familiar with a handful of APIs. Even then, assuming you knew how to get daily [OHLC data](https://en.wikipedia.org/wiki/Open-high-low-close_chart) for a crypto-asset like Bitcoin, in order to compare it to some other asset—say Amazon stock—you would have to eyeball trends from a website like Yahoo finance or scrape that data separately and build your own visualizations and metrics. In short, historical asset performance comparisons in the crypto space were difficult to conduct for all but the most technically savvy individuals, so I set out to build a tool that would remedy this,

In this post, I aim to describe a few key components of the dashboard, and also call out lessons learned from the process of iterating on the tool along the way. Prior to proceeding, I highly recommend that you read [the app’s README](https://github.com/pmaji/financial-asset-comparison-tool/blob/master/README.md) and take a look at [the UI](http://96.241.22.80:7597/) and [code base itself](https://github.com/pmaji/financial-asset-comparison-tool/blob/master/app.R), as this will provide the context necessary to understanding the rest of the commentary below.

I’ll start by delving into a few principles that I find to be to key when designing analytic dashboards, drawing on the asset comparison dashboard as my exemplar, and will end with some discussion of the relative utility of a few packages integral to the app. Overall, my goal is not to focus on the tool that I built alone, but to highlight a few main best practices when it comes to building dashboards for any analysis.

Build the app around the story, not the other way around.

Before ever writing a single line of code for an analytic app, I find that it is absolutely imperative to have a clear vision of the story that the tool must tell. I do not mean by this that you should already have conclusions about your data that you will then force the app into telling, but rather, that you must know *how* you want your user to interact with the app in order glean useful information.

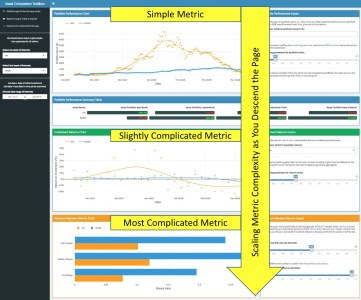
In the case of my asset comparison tool, I wanted to serve multiple audiences—everyone from a casual trader who just wanted to see which investment produced the greatest net profit over a period of time, to a more experience trader, who had more nuanced questions about risk-adjusted return on investment given varying discount rates. The trick is thus building the app in such a way that serves all possible audiences without hindering any one type of user in particular.

The way I designed my app to meet this need was to build the UI such that as you descend the various sections vertically, the metrics displayed scale in complexity. My reasoning for this becomes apparent when you consider the two extremes in terms of users—the most basic vs. the most advanced trader.

The most basic user will care only about the assets of interest, the time period they want to examine, and how their initial investment performed over time. As such, they will start with the sidebar, input their assets and time frame of choice, and then use the top right-most input box to modulate their initial investment amount (although some may choose to stick with the default value here). They will then see the first chart change to reflect their choices, and they will see, both visually, and via the summary table below, which asset performed better.

The experienced trader, on the other hand, will start off exactly as the novice did, by choosing assets of interest, a time frame of reference, and an initial investment amount. They may then choose to modulate the [LOESS](https://en.wikipedia.org/wiki/Local_regression) parameters as they see fit, descending the page, looking over the simple returns section, perhaps stopping to make changes to the corresponding inputs there, and finally ending at the bottom of the page—at the Sharpe Ratio visualizations. Here they will likely spend more time—playing around with the time period over which to measure returns and changing the risk-free rate to align with their own personal macroeconomic assumptions.

The point of these two examples is to illustrate that the app by dint of its structure alone guides the user through the analytic story in a waterfall-like manner—building from simple portfolio performance, to relative performance, to the most complicated metrics for risk-adjusted returns. This keeps the novice trader from being overwhelmed or confused, and also allows the most experienced user to follow the same line of thought that they would anyway when comparing assets, while following a logical progression of complexity, as shown via the screenshot below.



Once you think you have a structure that guides all users through the story you want them to experience, test it by asking yourself if the app flows in such a way that you could pose and answer a logical series of questions as you navigate the app without any gaps in cohesion. In the case of this app, the questions that the UI answers as you descend are as follows:

* How do these assets compare in terms of absolute profit?
* How do these assets compare in terms of simple return on investment?
* How do these assets compare in terms of variance-adjusted and/or risk-adjusted return on investment?

Thus, when you string these questions together, you can make statements of the type: “Asset X seemed to outperform Asset Y in terms of absolute profit, and this trend held true as well when it comes to simple return on investment, over varying time frames. That said, when you take into account the variance inherent to Asset X, it seems that Asset Y may have been the best choice, as the excess downside risk associated with Asset X outweighs its excess net profitability.

Too many cooks in the kitchen—the case for a functional approach to app-building.

While the design of the UI of any analytic app is of great importance, it’s important to not forget that the code base itself should also be well-designed; a fully-functional app from the user’s perspective can still be a terrible app to work with if the code is a jumbled, incomprehensible mess. A poorly designed code base makes QC a tiresome, aggravating process, and knowledge sharing all but impossible.

For this reason, I find that sourcing a separate R script file containing all analytic functions necessitated by the app is the best way to go, as done below.

# source the Functions.R file, where all analytic functions for the app are stored

source("Functions.R")

Not only does this allow for a more comprehensible and less-cluttered App.R, but it also drastically improves testability and reusability of the code. Consider the example function below, used to create the portfolio performance chart in the app (first box displayed in the UI, center middle).

build\_portfolio\_perf\_chart <- function(data, port\_loess\_param = 0.33){

port\_tbl <- data[,c(1,4:5)]

# grabbing the 2 asset names

asset\_name1 <- sub('\_.\*', '', names(port\_tbl)[2])

asset\_name2 <- sub('\_.\*', '', names(port\_tbl)[3])

# transforms dates into correct type so smoothing can be done

port\_tbl[,1] <- as.Date(port\_tbl[,1])

date\_in\_numeric\_form <- as.numeric((port\_tbl[,1]))

# assigning loess smoothing parameter

loess\_span\_parameter <- port\_loess\_param

# now building the plotly itself

port\_perf\_plot <- plot\_ly(data = port\_tbl, x = ~port\_tbl[,1]) %>%

# asset 1 data plotted

add\_markers(y =~port\_tbl[,2],

marker = list(color = '#FC9C01'),

name = asset\_name1,

showlegend = FALSE) %>%

add\_lines(y = ~fitted(loess(port\_tbl[,2] ~ date\_in\_numeric\_form, span = loess\_span\_parameter)),

line = list(color = '#FC9C01'),

name = asset\_name1,

showlegend = TRUE) %>%

# asset 2 data plotted

add\_markers(y =~port\_tbl[,3],

marker = list(color = '#3498DB'),

name = asset\_name2,

showlegend = FALSE) %>%

add\_lines(y = ~fitted(loess(port\_tbl[,3] ~ date\_in\_numeric\_form, span = loess\_span\_parameter)),

line = list(color = '#3498DB'),

name = asset\_name2,

showlegend = TRUE) %>%

layout(

title = FALSE,

xaxis = list(type = "date",

title = "Date"),

yaxis = list(title = "Portfolio Value ($)"),

legend = list(orientation = 'h',

x = 0,

y = 1.15)) %>%

add\_annotations(

x= 1,

y= 1.133,

xref = "paper",

yref = "paper",

text = "",

showarrow = F

)

return(port\_perf\_plot)

}

Writing this function in the sourced Functions.R file instead of directly within the App.R allows for the developer to first test the function itself with fake data—i.e. data not gleaned from the reactive inputs. Once it has been tested in this way, it can be integrated in the app.R on the server side as shown below, with very little code.

output$portfolio\_perf\_chart <-

debounce(

renderPlotly({

data <- react\_base\_data()

build\_portfolio\_perf\_chart(data, port\_loess\_param = input$port\_loess\_param)

}),

millis = 2000) # sets wait time for debounce

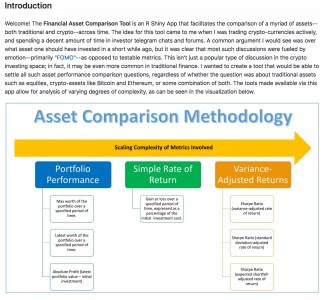
This process allows for better error-identification and troubleshooting. If, for example, you want to change the work accomplished by the analytic function in some way, you can make the changes necessary to the code, and if the app fails to produce the desired outcome, you simply restart the chain: first you test the function in a vacuum outside of the app, and if it runs fine there, then you know that you have a problem with the way the reactive inputs are integrating with the function itself. This is a huge time saver when debugging.

Lastly, this allows for ease of reproducibility and hand-offs. If, say, one of your functions simply takes in a dataset and produces a chart of some sort, then it can be easily copied from the Functions.R and reused elsewhere. I have done this too many times to count, ripping code from project and, with a few alterations, instantly applying it in other contexts. This is easy to do if the functions are written in a manner not dependent on a particular Shiny reactive structure. For all these reasons, it makes sense in most cases to keep the code for the app UI and inputs cleanly separated from the analytic functions via a sourced R script.

Dashboard documentation—both a story and a manual, not one or the other.

When building an app for a customer at work, I never simply write an email with a link in it and write “here you go!” That will result in, at best, a steep learning curve, and at worst, an app used in an unintended way, resulting in user frustration or incorrect results. I always meet with the customer, explain the purpose and functionalities of the tool, walk through the app live, take feedback, and integrate any key takeaways into further iterations.

Even if you are just planning on writing some code to put up on GitHub, you should still consider all of these steps when working on the documentation for your app. In most cases, the README is the epicenter of your documentation—the README is your meeting with the customer. I always start my READMEs with a high-level introduction to the purpose of the app—hopefully written or supplemented with visuals (as seen below) that are easy to understand and will capture the attention of browsing passers-by.



In some cases apps are meant to answer one particular question, and might have a variety of filters or supplemental functionalities. As can be seen, in that README, I spend a great deal of time on the methodology after making the overall purpose clear, calling out additional options along the way. In the case of the README for the Asset Comparison Tool, however, the story is a bit different. Given that there are many questions that the app seeks to answer, it makes sense to answer each in turn, writing the README in such a way that its progression mirrors the logical flow of the progression intended for the app’s user.

One should of course not neglect to cover necessary technical information in the README as well. Anything that is not immediately clear from using the app should be clarified in the README—from calculation details to the source of your data, etc. Finally, don’t neglect the iterative component! Mention how you want to interact with prospective users and collaborators in your documentation. For example, I normally call out how I would like people to use the Issues tab on GitHub to propose any changes or additions to the documentation, or the app in general. In short, your documentation must include both the story you want to tell, and a manual for your audience to follow.

Why Shiny Dashboard?

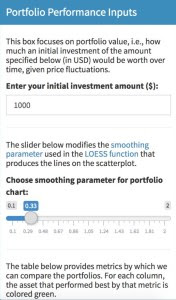
Shiny Dashboard provides the biggest bang for your buck in terms of how much UI complexity and customizability you get out of just a small amount of code.

I can think of few cases where any analyst or developer would prefer longer, more verbose code to a shorter, succinct solution. That said, Shiny Dashboard’s simplicity when it comes to UI manipulation and customization is not just helpful because it saves you time as a coder, but because it is intuitive from the perspective of your audience.

Most of the folks that use the tools I have built to shed insight into economic questions don’t know how to code in R or Python, but they can, with a little help from extensive commenting and detailed documentation, understand the broad structure of an app coded in Shiny Dashboard format. This is, I believe, largely a function of two features of Shiny Dashboard: the colloquial-English-like syntax of the code for UI elements, and the lack of the necessity for in-line or external CSS.

As you can see from the example below, Shiny Dashboard’s [system of “boxes”](https://rstudio.github.io/shinydashboard/structure.html#boxes) for UI building is easy to follow. Users can see a box in the app and easily tie that back to a particular box in the UI code.

Here is the box as visible to the user:



And here is the code that produces the box:

box(

title = "Portfolio Performance Inputs",

status= "primary",

solidHeader = TRUE,

h5("This box focuses on portfolio value, i.e., how much an initial investment of the amount specified below (in USD) would be worth over time, given price fluctuations."),

textInput(

inputId = "initial\_investment",

label = "Enter your initial investment amount ($):",

value = "1000"),

hr(),

h5("The slider below modifies the", a(href = "https://stats.stackexchange.com/questions/2002/how-do-i-decide-what-span-to-use-in-loess-regression-in-r", "smoothing parameter"), "used in the", a(href = "https://en.wikipedia.org/wiki/Local\_regression", "LOESS function"), "that produces the lines on the scatterplot."),

sliderInput(

inputId = "port\_loess\_param",

label = "Smoothing parameter for portfolio chart:",

min = 0.1,

max = 2,

value = .33,

step = 0.01,

animate = FALSE

),

hr(),

h5("The table below provides metrics by which we can compare the portfolios. For each column, the asset that performed best by that metric is colored green."),

height = 500,

width = 4

)

Secondly, and somewhat related to the first point, with Shiny Dashboard, much of the coloring and overall UI design comes pre-made via dashboard-wide “[skins](https://rstudio.github.io/shinydashboard/appearance.html#skins)”, and box-specific “[statuses](https://rstudio.github.io/shinydashboard/appearance.html#statuses-and-colors).”

This is great if you are okay sacrificing a bit of control for a significant reduction in code complexity. In my experience dealing with non-coding-proficient audiences, I find that in-line CSS or complicated external CSS makes folks far more uncomfortable with the code in general. Anything you can do to reduce this anxiety and make those using your tools feel as though they understand them better is a good thing, and Shiny Dashboard makes that easier.

Shiny Dashboard’s combination of sidebar and boxes makes for easy and efficient data processing when your app has a waterfall-like analytic structure.

Having written versions of this app both in base Shiny and using Shiny Dashboard, the number one reason I chose Shiny Dashboard was the fact that the analytic questions I sought to solve followed a waterfall-like structure, as explained in the previous section. This works perfectly well with Shiny Dashboard’s combination of [sidebar input controls](https://rstudio.github.io/shinydashboard/structure.html#sidebar) and inputs within UI boxes themselves.

The inputs of primordial importance to all users are included in the sidebar UI: the two assets to analyze, and the date range over which to compare their performance. These are the only inputs that all users, regardless of experience or intent, must absolutely use, and when they are changed, all views in the dashboard will be affected. All other inputs are stored in the UI Boxes adjacent to the views that they modulate. This makes for a much more intuitive and fluid user experience, as once the initial sidebar inputs have been modulated, the sidebar can be hidden, as all other non-hidden inputs affect only the visualizations to which they are adjacent.

This waterfall-like structure also makes for more efficient reactive processes on the Shiny back-end. The inputs on the sidebar are parameters that, when changed, force the main reactive function that creates that primary dataset to fire, thus recreating the base dataset (as can be seen in the code for that base datasets creation below).

# utility functions to be used within the server; this enables us to use a textinput for our portfolio values

exists\_as\_number <- function(item) {

!is.null(item) && !is.na(item) && is.numeric(item)

}

# data-creation reactives (i.e. everything that doesn't directly feed an output)

# first is the main data pull which will fire whenever the primary inputs (asset\_1a, asset\_2a, initial\_investment, or port\_dates1a change)

react\_base\_data <- reactive({

if (exists\_as\_number(as.numeric(input$initial\_investment)) == TRUE) {

# creates the dataset to feed the viz

return(

get\_pair\_data(

asset\_1 = input$asset\_1a,

asset\_2 = input$asset\_2a,

port\_start\_date = input$port\_dates1a[1],

port\_end\_date = input$port\_dates1a[2],

initial\_investment = (as.numeric(input$initial\_investment))

)

)

} else {

return(

get\_pair\_data(

asset\_1 = input$asset\_1a,

asset\_2 = input$asset\_2a,

port\_start\_date = input$port\_dates1a[1],

port\_end\_date = input$port\_dates1a[2],

initial\_investment = (0)

)

)

}

})

Each of the visualizations are then produced via their own separate reactive functions, each of which takes as an input the main reactive (as shown below). This makes it so that whenever the sidebar inputs are changed, all reactives fire and all visualizations are updated; however, if all that is changed is a single loess smoothing parameter input, only the reactive used in the creation of that particular parameter-dependent visualization fires, which makes for great computational efficiency.

# Now the reactives for the actual visualizations

output$portfolio\_perf\_chart <-

debounce(

renderPlotly({

data <- react\_base\_data()

build\_portfolio\_perf\_chart(data, port\_loess\_param = input$port\_loess\_param)

}),

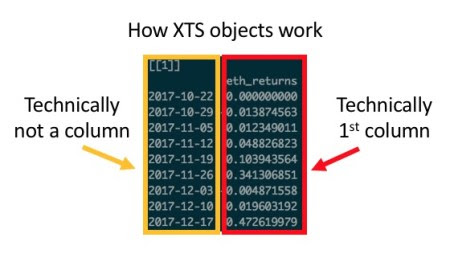
millis = 2000) # sets wait time for debounce

Why Plotly?

Plotly vs. ggplot is always a fun subject for discussion among folks who build visualizations in R. Sometimes I feel like such discussions just devolve into the same type of argument as R vs. Python for data science (my answer to this question being just pick one and learn it well), but over time I have found that there are actually some circumstances where the plotly vs. ggplot debate can yield cleaner answers.

In particular, I have found in working on this particular type of analytic app that there are two areas where plotly has a bit of an advantage: clickable interactivity, and wide data.

Those familiar with ggplot will know that every good ggplot begins with long data. It is possible, via some functions, but that transformation can sometimes be problematic. While there are essentially no circumstances in which it is impossible to transform wide data into long format, there are a handful of cases where it is excessively cumbersome: namely, when dealing with indexed xts objects (as shown below) or time series / OHLC-styled data.



In these cases—either due to the sometimes-awkward way in which you have to handle rowname indexes in xts, or the time and code complexity saved by not having to transform every dataset into long format—plotly offers efficiency gains relative to ggplot.

The aforementioned efficiency gains are a reason to choose plotly in some cases because it makes the life of the coder easier, but there are also reasons why it sometimes make the life of the user easier as well.

If one of the primary utilities of a visualization is to allow the user the ability to seamlessly and intuitively zoom in on, select, or filter the data displayed, particularly in the context of a Shiny App, then plotly should be strongly considered. Sure, [ggplotly wrappers](https://www.rdocumentation.org/packages/plotly/versions/4.7.1/topics/ggplotly?tap_a=5644-dce66f&tap_s=10907-287229) can be used to make a ggplot interactive, but with an added layer of abstraction comes an added layer of possible errors. While in most cases a ggplotly wrapper should work seamlessly, I have found that, particularly in cases where auto-sizing and margin size specification is key, ggplotly can require a great deal of added code in order to work correctly in a Shiny context.

In summary, when considering when to start with plotly vs. when to start with ggplot, I find one question to be particularly helpful: what do I value most—visual complexity and/or customization, or interactive versatility and/or preserving wide data?

If I choose the former, then ggplot is what I need; otherwise, I go with plotly. More often than not I find that ggplot emerges victorious, but even if you disagree with me in my decision-making calculus, I think it is helpful to at least think through what your personal calculus is. This will save you time when coding, as instead of playing around with various types of viz, you can simply pose the question(s) behind your calculus and know quickly what solution best fits your problem.

Why Formattable?

The case for [formattable](https://github.com/renkun-ken/formattable) is, in my opinion, a much easier case to make than arguing for plotly vs. ggplot. The only question worth asking when deciding on whether or not to use formattable in your app is: do I want my table to tell a quick story via added visual complexity within the same cell that contains my data, or is a reference table all I am looking for? If you chose the former, formattable is probably a good way to go. You’ll notice as well that the case for formattable is very specific–in most cases there is likely a simpler solution via the [DT](https://rstudio.github.io/DT/) or [kableExtra](https://cran.r-project.org/web/packages/kableExtra/index.html) packages.

The one downside that I have encountered in dealing with formattable code is the amount of code necessary to generate even moderately complicated tables. That said, this problem is easily remedied via a quick function that we can use to kill most of the duplicative coding, as seen in the example below.

First, here is the long form version:

react\_formattable <- reactive({

return(

formattable(react\_port\_summary\_table(),

list(

"Asset Portfolio Max Worth" = formatter("span",

style = x ~ style(

display = "inline-block",

direction = "rtl",

"border-radius" = "4px",

"padding-right" = "2px",

"background-color" = csscolor("darkslategray"),

width = percent(proportion(x)),

color = csscolor(gradient(x, "red", "green"))

)),

"Asset Portfolio Latest Worth" = formatter("span",

style = x ~ style(

display = "inline-block",

direction = "rtl",

"border-radius" = "4px",

"padding-right" = "2px",

"background-color" = csscolor("darkslategray"),

width = percent(proportion(x)),

color = csscolor(gradient(x, "red", "green"))

)),

"Asset Portfolio Absolute Profit" = formatter("span",

style = x ~ style(

display = "inline-block",

direction = "rtl",

"border-radius" = "4px",

"padding-right" = "2px",

"background-color" = csscolor("darkslategray"),

width = percent(proportion(x)),

color = csscolor(gradient(x, "red", "green"))

)),

"Asset Portfolio Rate of Return" = formatter("span",

style = x ~ style(

display = "inline-block",

direction = "rtl",

"border-radius" = "4px",

"padding-right" = "2px",

"background-color" = csscolor("darkslategray"),

width = percent(proportion(x)),

color = csscolor(gradient(x, "red", "green"))

))

)

)

)

})

This code can easily be shortened via the integration of a custom function, as shown below.

simple\_formatter <- function(){

formatter("span",

style = x ~ style(

display = "inline-block",

direction = "rtl",

"border-radius" = "4px",

"padding-right" = "2px",

"background-color" = csscolor("darkslategray"),

width = percent(proportion(x)),

color = csscolor(gradient(x, "red", "green"))

))

}

react\_formattable <- reactive({

return(

formattable(react\_port\_summary\_table(),

list(

"Asset Portfolio Max Worth" = simple\_formatter(),

"Asset Portfolio Latest Worth" = simple\_formatter(),

"Asset Portfolio Absolute Profit" = simple\_formatter(),

"Asset Portfolio Rate of Return" = simple\_formatter()

)

)

)

})

As can be seen, formattable allows for a great deal of added complexity in crafting your table—complexity that may not be suited for all apps. That said, if you do want to quickly draw a user’s attention to something in a table, formattable is a great solution, and most of the details of the code can be greatly simplified via a function, as shown.

Conclusions:

That was a lot—I know—but I hope that from this commentary and my exemplar of the [Asset Comparison Tool](https://github.com/pmaji/financial-asset-comparison-tool) more generally has helped to inform your understanding of how dashboards can serve as a helpful analytic tool. Furthermore, I hope to have prompted some thoughts as to the best practices to be followed when building such a tool. I’ll end with a quick *tl;dr*:

* Shave complexity wherever possible, and make code as simple as possible by keeping the code for the app’s UI and inner mechanism (inputs, reactives, etc.) separate from the code for the analytic functions and visualizations.
* Build with the most extreme cases in mind (think of how your most edge-case user might use the app, and ensure that behavior won’t break the app)
* Document, document, and then document some more. Make your README both a story and a manual.
* Give Shiny Dashboard a shot if you want an easy-to-construct UI over which you don’t need complete control when it comes to visual design.
* Pick your visualization packages based on what you want to prioritize for your user, not the other way around (this applies to ggplot, plotly, formattable, etc.).