

I wish this post existed when I was struggling to add interactive plots to my Shiny app. I was mainly focused on recreating functionality found in other “dashboarding” applications. When looking for options, I found that [htmlwidgets](https://www.htmlwidgets.org/) were the closest to what companies usually expect. However, while they are great for client-side interactivity, I often hit walls with them when I try to add click-through interactivity because the functionality is either not there, is very limited, or is bloated. With r2d3 there is more work, but the gains in customization and interactivity make it by far the best choice, in my opinion.

I asked a good friend at work to help me test the sample app provided in this post. She was able to run it easily, but then told me that she didn’t know that she was supposed to click on things. Adding interactive plots is one of the most important capabilities to include in a Shiny app. Sadly though, it seems that very few do it. If we wish to offer an alternative to enterprise reporting and BI tools by using Shiny, we need to do our best to match the interactivity those other tools seem to offer out of the box.

**The sample app**

I put together a sample app that should run in your R session by simply copying the code. This will allow us to focus on the details of the approach, and not on the setup.

A working version of the app is available here: [Shiny-r2d3-app](https://beta.rstudioconnect.com/content/3940/)

In this app, we can click on the bars and see the DT object update based on the value of the bar. When the drop-down changes, the plot will update with a nice transition, as well.

**“D3 is hard”**

The title is a quote of a luminary in the R community. A few months ago, I told him that I wanted to start using r2d3 but was struggling with making heads or tails of D3. This person has forgotten more than I will ever learn about pretty much any subject. If he says it’s hard, then I’m in for a world of hurt. Nevertheless, my naivete and stubbornness prevailed.

I’ve since discovered that D3 is a language with which the desired result can be obtained by using one of several coding approaches. The more I learn to use it, the more I like its flexibility as a stand-alone visualization language.

One thing that helped was to realize that D3 and ggplot2 are similar in the amount of flexibility they offer. Picture that what you are drawing for a bar plot are the actual rectangles, almost as if you’re using geom\_rect(). Except that in D3, the 0,0 coordinates are top/left, as opposed to bottom/left, so we have to flip our thinking upside down when we create a visualization with D3. In addition, the vertical and horizontal positions and sizes are expressed in fractions (read: percentages), so there are no absolute positions.

**A good way to start**

After trying out several approaches, I think that a good way to start is by having a few “primer” D3 scripts that can be modified to suit a particular app.

r2d3 calls a D3 script with a .js extension. As a result, the D3 code sits outside the R script, away from view. With r2d3, a data.frame can be used to pass all sorts of attributes (x/y coordinates, colors, etc.) to D3.

A good way of thinking about these “primers” is that you are building your own geoms as .js scripts. So, once it’s done, you can pass the regular “right-side-up” coordinate data to r2d3 and it will know how to calculate the proper offsets to place the shapes in the correct spot.

**A first primer**

The idea in this section is to provide the smallest possible example that covers what I feel are the most important pieces that make up a presentable and functional product. My hope is that, if you find this interesting and useful for your line of work, you will take your time to dissect what each code section does, to learn the principles of this approach. This way, you can customize and even expand on the primer.

The first example below is not the full primer. Instead, it is the section where most of the nuances of how the primer works exist. I’ll use that to explain some of the mechanics.

You can copy-paste the following code in your R session and run it without worrying about file dependencies. I know how important that is when learning new things, so I’m using a small workaround to providing r2d3 a separate .js file by saving the contents of a character variable that contains the D3 script into a temporary file. This is probably not something that you’ll do in a final Shiny app, but it works well for this example. Based on how the R Views’ code highlighter is setup, all of the D3 code will be in red, and the R code mostly in black:

library(shiny)

library(dplyr)

library(r2d3)

library(forcats)

# D3 code inside an R character variable

r2d3\_script <- "

// !preview r2d3 data= data.frame(y = 0.1, ylabel = '1%', fill = '#E69F00', mouseover = 'green', label = 'one', id = 1)

function svg\_height() {return parseInt(svg.style('height'))}

function svg\_width() {return parseInt(svg.style('width'))}

function col\_top() {return svg\_height() \* 0.05; }

function col\_left() {return svg\_width() \* 0.20; }

function actual\_max() {return d3.max(data, function (d) {return d.y; }); }

function col\_width() {return (svg\_width() / actual\_max()) \* 0.55; }

function col\_heigth() {return svg\_height() / data.length \* 0.95; }

var bars = svg.selectAll('rect').data(data);

bars.enter().append('rect')

.attr('x', col\_left())

.attr('y', function(d, i) { return i \* col\_heigth() + col\_top(); })

.attr('width', function(d) { return d.y \* col\_width(); })

.attr('height', col\_heigth() \* 0.9)

.attr('fill', function(d) {return d.fill; })

.attr('id', function(d) {return (d.label); })

.on('click', function(){

Shiny.setInputValue('bar\_clicked', d3.select(this).attr('id'), {priority: 'event'});

})

.on('mouseover', function(){

d3.select(this).attr('fill', function(d) {return d.mouseover; });

})

.on('mouseout', function(){

d3.select(this).attr('fill', function(d) {return d.fill; });

});

"

# Save D3 code into a tempfile

r2d3\_file <- tempfile()

writeLines(r2d3\_script, r2d3\_file)

# Shiny app starts here

ui <- fluidPage(

d3Output("d3")

)

server <- function(input, output, session) {

output$d3 <- renderD3({

gss\_cat %>%

group\_by(marital) %>%

tally() %>%

arrange(desc(n)) %>%

mutate(

y = n,

ylabel = prettyNum(n, big.mark = ","),

fill = "#E69F00",

mouseover = "#0072B2"

) %>%

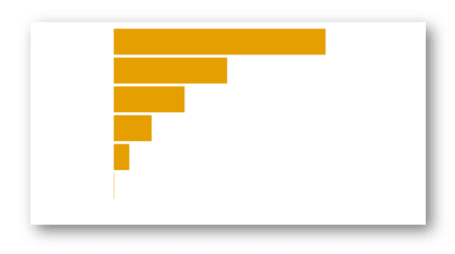
r2d3(r2d3\_file)

# ^^ Use the temp file containing the D3 code

})}

shinyApp(ui = ui, server = server)

The result should look like the screenshot below. In your R session, hovering over the bar will change the color. Also notice that the bars do not cover the entire window. This is because there are limits placed in the way of ratios within the functions used on the top of the script.



**Code breakdown**

**First, is the D3 code:**

* I start by defining some canvas size function beginning with: function svg\_height() {return parseInt(svg.style('height'))}. These allow for the correct relative placement and size, as well as adapting to a window resize. For example: function actual\_max() {return d3.max(data, function (d) {return d.y; }); } obtains the value of the longest bar, and then: function col\_width() {return (svg\_width() / actual\_max()) \* 0.55; } makes sure that the largest rectangle (representing a bar) drawn is 55% the size of the window. I used to define these as regular D3 variables, but found that as functions, they worked more consistently when running with Shiny.
* With var bars = svg.selectAll('rect').data(data);, we create a new rectangle – better said, a new rectangle set. Just like with geom\_rect(), if you pass a vector with multiple values, it will create multiple rectangles. The last function, data(), tells D3 to use the data data set, which is the default name that r2d3 is using when it translates our data.frame to a D3-friendly format. This is the “secret sauce” that allows us to use that data as attributes of the plot.
* The rectangles are initially drawn with: bars.enter().append('rect'). This will work fine as long as nothing changes. But with Shiny, we want change, so in a later section, I will introduce the bars.transition() function.
* Next, are the attributes (.attr). Attributes are interesting in these kinds of objects. They are all named as a character variable (x, fill, etc.), so it’s essentially free-form. Each type of D3 shape has its own set of expected attributes, such as x, y, and width, but I can also pass a “made-up” attribute and the script will not fail. In other words, if you pass an attribute of a “reserved” name for the shape. it will be used; for example, r is the attribute for radius of a D3 circle. But if the attribute does not exist, it just becomes metadata that we can use later on if we want. This comes in handy if we want an ID field to be passed to Shiny, but that ID field is not displayed in the plot. The downside is that a misspelled attribute will fail silently, so it makes debugging a bit difficult. In other words, make sure that your attributes are spelled correctly! In the meantime, defining x is easy because we want it to be as far to the left as possible.
* Most attributes are set based on data passed via r2d3. We do that by wrapping the value of the attribute inside a function. We already told D3 where the data comes from, so it is implied that in function(d) the data object will be represented by d. Another interesting thing about these functions is the second argument, usually represented by i. It represents the “row number” of the observation. This means that a function like function(d, i) { return d.x \* i} will give the attribute the value of the x variable of the data.frame we passed to r2d3, times the row number. So .attr('fill', function(d) {return d.fill; }) simply passes the fill value of our data.frame to D3. Notice that we can name these fields whatever we want; we just need to map them appropriately. With a primer, I found that it’s better to keep either matching (or at the very least, generic) names so we can use them for other plots.
* The on() functions track named events, such as click, mouseover, and mouseout.
* The click function will use a Shiny JavaScript function that makes the interaction possible. In Shiny.setInputValue('bar\_clicked', d3.select(this).attr('id'), {priority: 'event'});, I specify the name of the input inside Shiny, so bar\_clicked becomes input$bar\_clicked in R. The attribute id is the value passed to R via that input. This is only a brief introduction to the topic; a much more detailed explanation with illustrations can be found in the [r2d3 site](https://rstudio.github.io/r2d3/articles/shiny.html#d3-to-shiny).
* The mouseover and mouseout events are used to get the color-changing, hover-over effect. On mouseover, the fill attribute is updated to use the highlighting color and then restore it to the original color when the pointer leaves with mouseout.

**For the R/Shiny code:**

* As mentioned above, using r2d3\_file <- tempfile() and then writeLines(r2d3\_script, r2d3\_file) is done to keep the D3 and R code in one location. This allows you to copy and run the script without worrying about dependencies.
* r2d3 includes functions to interact with Shiny. The d3Output() function is used in the ui section of the app, and renderD3() is used in the server section of the app.
* Using dplyr, the forcats::gss\_cat data is transformed to fit what the primer expects. In other words, the variable that the total count obtained with tally() is renamed to y. Additionally, new fields are added to specify the colors. A note about colors with D3: you can pass color names (“red”), or the Hex code of the color (“#E69F00”). Some additional tips for Hex color selection can be found in the [ggplot2 cookbook](http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/#a-colorblind-friendly-palette). A very nice application to test different color schemes and explore contrast with different color deficiencies is [here](http://projects.susielu.com/viz-palette).
* Thanks to the fact that the r2d3() function uses the data as its first argument, we can simply pipe (%>%) the dplyr transformations directly to it. The only argument to pass to r2d3() is the location of the new temporary file.

**The full example**

Here is the full code for the sample app linked above. The D3 script is what I would consider a more complete “primer” that you can use in other apps. Copy and run the code to try out the Shiny app; as mentioned before, it should run without having to worry about any other file dependencies. More explanation and code breakdown is available after this code section:

library(shiny)

library(dplyr)

library(r2d3)

library(forcats)

library(DT)

library(rlang)

r2d3\_script <- "

// !preview r2d3 data= data.frame(y = 0.1, ylabel = '1%', fill = '#E69F00', mouseover = 'green', label = 'one', id = 1)

function svg\_height() {return parseInt(svg.style('height'))}

function svg\_width() {return parseInt(svg.style('width'))}

function col\_top() {return svg\_height() \* 0.05; }

function col\_left() {return svg\_width() \* 0.20; }

function actual\_max() {return d3.max(data, function (d) {return d.y; }); }

function col\_width() {return (svg\_width() / actual\_max()) \* 0.55; }

function col\_heigth() {return svg\_height() / data.length \* 0.95; }

var bars = svg.selectAll('rect').data(data);

bars.enter().append('rect')

.attr('x', col\_left())

.attr('y', function(d, i) { return i \* col\_heigth() + col\_top(); })

.attr('width', function(d) { return d.y \* col\_width(); })

.attr('height', col\_heigth() \* 0.9)

.attr('fill', function(d) {return d.fill; })

.attr('id', function(d) {return (d.label); })

.on('click', function(){

Shiny.setInputValue('bar\_clicked', d3.select(this).attr('id'), {priority: 'event'});

})

.on('mouseover', function(){

d3.select(this).attr('fill', function(d) {return d.mouseover; });

})

.on('mouseout', function(){

d3.select(this).attr('fill', function(d) {return d.fill; });

});

bars.transition()

.duration(500)

.attr('x', col\_left())

.attr('y', function(d, i) { return i \* col\_heigth() + col\_top(); })

.attr('width', function(d) { return d.y \* col\_width(); })

.attr('height', col\_heigth() \* 0.9)

.attr('fill', function(d) {return d.fill; })

.attr('id', function(d) {return d.label; });

bars.exit().remove();

// Identity labels

var txt = svg.selectAll('text').data(data);

txt.enter().append('text')

.attr('x', width \* 0.01)

.attr('y', function(d, i) { return i \* col\_heigth() + (col\_heigth() / 2) + col\_top(); })

.text(function(d) {return d.label; })

.style('font-family', 'sans-serif');

txt.transition()

.duration(1000)

.attr('x', width \* 0.01)

.attr('y', function(d, i) { return i \* col\_heigth() + (col\_heigth() / 2) + col\_top(); })

.text(function(d) {return d.label; });

txt.exit().remove();

// Numeric labels

var totals = svg.selectAll().data(data);

totals.enter().append('text')

.attr('x', function(d) { return ((d.y \* col\_width()) + col\_left()) \* 1.01; })

.attr('y', function(d, i) { return i \* col\_heigth() + (col\_heigth() / 2) + col\_top(); })

.style('font-family', 'sans-serif')

.text(function(d) {return d.ylabel; });

totals.transition()

.duration(1000)

.attr('x', function(d) { return ((d.y \* col\_width()) + col\_left()) \* 1.01; })

.attr('y', function(d, i) { return i \* col\_heigth() + (col\_heigth() / 2) + col\_top(); })

.attr('d', function(d) { return d.x; })

.text(function(d) {return d.ylabel; });

totals.exit().remove();

"

r2d3\_file <- tempfile()

writeLines(r2d3\_script, r2d3\_file)

ui <- fluidPage(

selectInput("var", "Variable",

list("marital", "rincome", "partyid", "relig", "denom"),

selected = "marital"),

d3Output("d3"),

DT::dataTableOutput("table"),

textInput("val", "Value", "Married")

)

server <- function(input, output, session) {

output$d3 <- renderD3({

gss\_cat %>%

mutate(label = !!sym(input$var)) %>%

group\_by(label) %>%

tally() %>%

arrange(desc(n)) %>%

mutate(

y = n,

ylabel = prettyNum(n, big.mark = ","),

fill = ifelse(label != input$val, "#E69F00", "red"),

mouseover = "#0072B2"

) %>%

r2d3(r2d3\_file)

})

observeEvent(input$bar\_clicked, {

updateTextInput(session, "val", value = input$bar\_clicked)

})

output$table <- renderDataTable({

gss\_cat %>%

filter(!!sym(input$var) == input$val) %>%

datatable()

})

}

shinyApp(ui = ui, server = server)

**Additions to D3 code**

Hopefully, you can see a coding pattern emerging in the more lengthy example above. Here are some explanations for items that are new or outside the pattern:

* The bars.transition() function “re-draws” the shape or text when the underlying data changes, when we make a change within the Shiny app. The duration() function defines the time that the changes take. Be sure to copy all of the attributes from the enter() function. This is needed when adding D3 plots into a Shiny app.
* The var txt = svg.selectAll('text').data(data); code adds a new text object, similar to geom\_text(). The same coding pattern as the rect shape applies. The additions are: a text() function that defines what its displayed on screen (note that there’s no attr('text',...), and the style() function to allow setting the font type size.

**Setting up the Shiny interactivity**

There are three options to integrate the Shiny input created inside the D3 script:

1. Have a given Shiny output react to the D3/Shiny input. An example would be to use it as a value to filter data in filter(id\_field == input$bar\_clicked). This works OK when there are not too many plots to integrate, but for a large dashboard, the second option would be better. An example of this approach can be found [here](https://rstudio.github.io/r2d3/articles/shiny.html#shiny-code).
2. Use Shiny’s observeEvent() to monitor the D3/Shiny input and have it run a specific action based on the value of the input. I usually use this approach to update another Shiny input in the app, and that is the approach used in this app.
3. Use the reactive() function to wrap all of the data transformations that are common across all of the plots inside the dashboard. Then have each plot use that function as the base of further dplyr transformations. That approach can be found in the [Enterprise Dashboards article](http://db.rstudio.com/best-practices/dashboards/) on db.rstudio.com; here is [a direct link to the code](https://github.com/sol-eng/db-dashboard/blob/f3f42eabe722207510e6670ad81e36722e0b3d44/local_app.R#L91-L116).

**Other R additions**

A few additional tips that are helpful, but not mandatory:

* To get the effect of keeping the selected bar with a different color than the others, I used an ifelse() inside the mutate() that checks if a particular row matches to the selected input: fill = ifelse(label != input$val, "#E69F00", "red").
* In this line: mutate(label = !!sym(input$var)), I am using rlang’s convention to allow for the plot to change the field that it is displaying. This is a very rare requirement in an app, so I hope that it doesn’t throw anyone off. This is an advanced R programming concept not necessary for D3/Shiny.
* I decided to use a separate field with the total count (y) and the label that will be shown in that bar (ylabel). It was easier for me to edit the format in R than in D3. Some may decide to do that in the D3 script.

**RStudio 1.2**

If you have the RStudio IDE Preview Release installed, you can easily preview the D3 visualization right in the Viewer pane. Information on how to do this is [here](https://rstudio.github.io/r2d3/#d3-preview).

In the first line in the script above, there is a D3 comment line with metadata that RStudio will pass to r2d3 so that you do not run R code in the console to see a preview. This integration also lets us use the IDE to edit the D3 file, which accelerates learning D3.

To try this out with the visualization above, copy and paste the contents of the r2d3\_script variable to a new D3 file inside the RStudio IDE.

**Closing words**

Thank you for making it this far! Even if you were just skimming, I hope one or two things I’ve shown were interesting enough to consider trying out the exercise.

Sometimes, we forget how far we have progressed on a subject and forget what it feels like to begin the learning process. Hopefully these explanations avoid this pitfall and will simplify your learning experience. Please feel free to ask questions or start a topic of discussion at [community.rstudio.com](https://community.rstudio.com/), where many are happy to help!

Here are some additional links to resources that you may want to check out. The first two I wrote for RStudio documentation:

* [Using r2d3 with Shiny](https://rstudio.github.io/r2d3/articles/shiny.html)
* [Enterprise-ready dashboards](http://db.rstudio.com/best-practices/dashboards/#using-r2d3-for-interactivity-and-drill-down)
* The [D3 API reference](https://github.com/d3/d3/blob/master/API.md) is really good, I use it often.
* The [r2d3 site](https://rstudio.github.io/r2d3/) has a great Gallery and articles to review. It has a section about [learning D3](https://rstudio.github.io/r2d3/articles/learning_d3.html).