You want to have a nice interactive visualization, e.g., in RShiny, where you can zoom in and out, subset the plot or even have hover and click effects? I guess your first shot would be plotly, and rightfully so. However, there is also a great alternative: rBokeh.

In particular, plotly does not have a function to draw rectangles. Just converting a ggplot object into plotly by ggplotly() does not allow for a properly customized callback.

In the course of this project, We learned a lot and grew to love rBokeh. Therefore, I want to share some ideas and solutions for some rather specific problems I came across. If you look for a general introduction to rBokeh,

**Write complex JavaScript callbacks**

An excellent quality of rBokeh is its capability to write customized callbacks with JavaScript (JS). We’re going to dive into this topic next:

My goal is to trigger something in my plot by hovering over different elements. For this showcase, I use the classic iris data again. When you hover over a point, you should see the corresponding species name in a highlighted box (check the plot!).

The complete code for this task is shown below but let me guide you through it step by step:

**Set up the information to trigger**

First, I set up the data to be triggered. This includes the names of the different iris species and where this „legend“ should appear in the plot.

my\_data <- tibble(name = unique(iris$Species),

x = c(2.1, 3.1, 4.1),

y = c(7.5, 8.0, 7.5))

As usual for an rBokeh plot, we need to call the figure() function first. Since we like to build a sort of legend by ourselves, I remove the legend by setting its location to NULL.  
The scatter plot is then initialized by ly\_points(). For visual purposes, I colorize the points according to their species and increase the point size. The next line, however, is quite essential for the functionality of my callback. You must define a name for the glyph (lname) in order to refer to it in the callback. Note that it is not necessary to call it like the glyph itself. I do this just for convenience.

plot <-

figure(data = iris,

legend\_location = NULL) %>%

ly\_points(x = Sepal.Width,

y = Sepal.Length,

color = Species,

size = 20,

lname = "points") %>%

The names of the species can be included in the plot by ly\_text(). We have to specify the data argument and overwrite the default data that was specified before in figure(). I specify the location and text arguments and align them appropriately. Furthermore, I set the color argument to transparent. This predefined CSS color ensures that the text is „hidden“ when the plot is first rendered and before we first hover over it. Of course, we have to define the lname argument again.

ly\_text(data = my\_data,

x = x,

y = y,

text = name,

align = "center",

baseline = "middle",

color = "transparent",

lname = "text") %>%

The centered rectangles are made by ly\_crect. The first special argument I set here is the alpha. Like for the ly\_text, I want the glyph transparent in the initial plot. However and strangely, some glyphs do not recognize transparent as a predefined color. The only remaining option is to set the alpha argument explicitly to zero, yet this requires an additional step later on. Nevertheless, we need to define some arbitrary color to change it later. Also, we must not forget to name the glyph!

ly\_crect(data = my\_data,

x = x,

y = y,

height = 0.5,

width = 1,

alpha = 0,

color = "blue",

lname = "rects")

That’s it with the initial plot. Now I need to find a linkage between each point in the scatter plot and its corresponding species in my\_data. This will heavily depend on the specific case you’re dealing with, but the following lapply does the job here. It returns a list in which each element refers to one line of the iris data and consequently to one point. Each list element is a list by itself and contains the position of the row in my\_data with the correct species. Since we use this information in our JS code, you have to account for the zero-based property of JS. Therefore, when we want to refer to the first line in my\_data, this is addressed by 0 and so on.  
One final note here: make sure that you store everything in lists since JS is not capable of digesting R vectors.

linkages <- lapply(iris$Species,

# To keep it general, I store all values in a list.

# This is not necessary if you link to just one value.

# However, whenever dealing with one-to-many relationships

# you need to store them in a list rather than in a vector

FUN = function(x) if(x == "setosa") list(0) # JS is zero-based!

else if (x == "versicolor") list(1)

else if (x == "virginica") list(2))

Now we finally arrive at the long-anticipated callback part. Since it should be initialized by hovering, we have to call tool\_hover and define which layer should trigger the callback (ref\_layer). In our case, its the scatter plot layer, of course.  
Since we want to include JS code, we need to call custom\_callback. In this, we can define which layers we want to have access to within JS (lnames) and even can make further R objects available inside of JS by the args argument. This must be a named list with the R object on the rhs and the object name you can address within JS on the lhs.

plot %>%

tool\_hover(

ref\_layer = "points",

callback = custom\_callback(

lnames = c("points", "rects", "text"),

args = list(links = linkages),

Let’s start with the actual JS code. In the first two lines, I invoke a debugger that makes it easier to develop and debug the code but more on this topic at the end of my post.  
To make the code more readable and reduce overhead, I define some variables. The first variable (indices) stores the hover information (cb\_data) or, more precisely, the indices of the currently hovered points. The further variables contain the underlying data of the respective glyphs.

code = paste0("

debugger;

console.log(points\_data.get('data'));

var indices = cb\_data.index['1d'].indices;

var pointsdata = points\_data.get('data');

var rectsdata = rects\_data.get('data');

var textdata = text\_data.get('data');",

This following code snippet is necessary because we had to set alpha = 0 within ly\_crect. Since we do not always want to hide this glyph, we have to overwrite and increase this value here. What happens is that when the plot is first called, alpha is zero. When we first hover the plot, alpha is set to 0.2. Note that we have to change both arguments, fill\_alpha, and line\_alpha. Now we would see the rectangles but change its color to transparent immediately. This is achieved in the next step.

"rects\_glyph.get('fill\_alpha').value = 0.2

rects\_glyph.get('line\_alpha').value = 0.2",

Each time the callback is triggered, that is the courser changes its position, the color for all text and rectangle elements is set to transparent. This is achieved by calling the JS rgba function (red, green, blue, alpha) and setting the alpha avalue to 0 for all glyphs. A simple for loop does this job here.  
For those who are not familiar with this kind of loop specification: we define an iterator variable i that’s initial value is zero. After each iteration, i is incremented by 1 (note that the ++ are behind i) as long as i is smaller than, e.g., rectsdata.fill\_color.length, that is the length of the vector.

"for (var i=0; i < rectsdata.fill\_color.length; i++){

rectsdata.line\_color[i] = 'rgba(255, 255, 255, 0)';

rectsdata.fill\_color[i] = 'rgba(255, 255, 255, 0)';

}

for (var i=0; i < textdata.text\_color.length; i++){

textdata.text\_color[i] = 'rgba(255, 255, 255, 0)';

}",

We are now approaching the core of the JS callback. These two loops actually trigger the behavior we desire. The outer loop is necessary if we hover over multiple points simultaneously. For each hovered point, its index position in the data set is stored in the variable ind. In the next step, we finally code the visible callback effect. For this, the element of the links list that corresponds to the currently hovered point is detected and its value is derived. This corresponds to the position in my\_data and then changes the color of the desired rectangle and text glyph. The inner loop is just included to keep my example as general as possible. In case that lists stored in links contains multiple elements by themselves.

"for(var i=0; i < indices.length; i++){

var ind = indices[i];

for (var j=0;j< links[ind].length; j++){

rectsdata.fill\_color[links[ind][j]] = '#0085AF';

rectsdata.line\_color[links[ind][j]] = '#0085AF';

textdata.text\_color[links[ind][j]] = '#013848';

}

}")

And that’s it! We built a fancy JS hover callback with rBokeh. And, as promised, here is the full code 

my\_data <- tibble(name = unique(iris$Species),

x = c(2.1, 3.1, 4.1),

y = c(7.5, 8.0, 7.5))

plot <-

figure(data = iris,

# Remove legend

legend\_location = NULL) %>%

ly\_points(x = Sepal.Width,

y = Sepal.Length,

# To check the correct functionality of the callback

color = Species,

# Increase size of the points

size = 20,

# IMPORTANT: define a name for the glyph!

lname = "points") %>%

ly\_text(data = my\_data,

x = x,

y = y,

text = name,

# Specify the correct alignment

align = "center",

baseline = "middle",

# IMPORTANT: Make the text invisible in the initial plot!

color = "transparent",

# IMPORTANT: define a name for the glyph!

lname = "text") %>%

ly\_crect(data = my\_data,

x = x,

y = y,

# Adapte the size of the rectangles

height = 0.5,

width = 1,

# IMPORTANT: make the rectangles transparent in the initial plot

alpha = 0,

# IMPORTANT: nevertheless, define an arbitrary color to refer to it in the

# callback

color = "blue",

# IMPORTANT: define a name for the glyph!

lname = "rects")

linkages <- lapply(iris$Species,

# To keep it general, I store all values in a list.

# This is not necessary if you link to just one value.

# However, whenever dealing with one-to-many relationships

# you need to store them in a list rather than in a vector

FUN = function(x) if(x == "setosa") list(0) # JS is zero-based!

else if (x == "versicolor") list(1)

else if (x == "virginica") list(2))

plot %>%

tool\_hover(

custom\_callback(

code = paste0("

debugger;

console.log(points\_data.get('data'));

var indices = cb\_data.index['1d'].indices;

var pointsdata = points\_data.get('data');

var rectsdata = rects\_data.get('data');

var textdata = text\_data.get('data');

rects\_glyph.get('fill\_alpha').value = 0.2

rects\_glyph.get('line\_alpha').value = 0.2

for (var i=0; i< rectsdata.fill\_color.length; i++){

rectsdata.line\_color[i] = 'rgba(255, 255, 255, 0)';

rectsdata.fill\_color[i] = 'rgba(255, 255, 255, 0)';

}

for (var i=0; i< textdata.text\_color.length; i++){

textdata.text\_color[i] = 'rgba(255, 255, 255, 0)';

}

for(var i=0; i < indices.length; i++){

var ind0 = indices[i];

for (var j=0;j< links[ind0].length; j++){

rectsdata.fill\_color[links[ind0][j]] = '#0085AF';

rectsdata.line\_color[links[ind0][j]] = '#0085AF';

textdata.text\_color[links[ind0][j]] = '#013848';

}

}

",

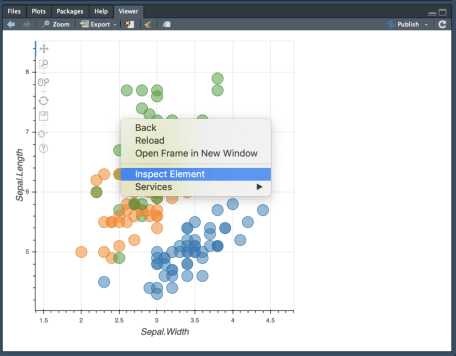
lnames = c("points", "rects", "text"),

args = list(links = linkages)),

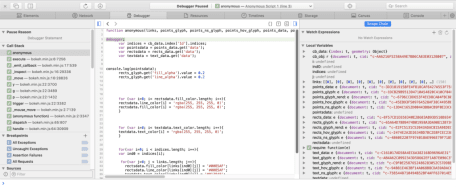
ref\_layer = "points")

**Final remarks on debugging**

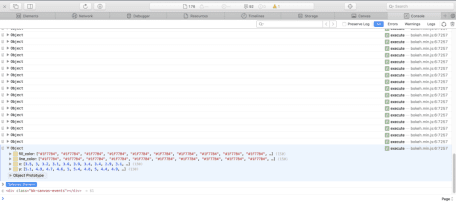
As mentioned above, it is advisable to include a debugger; in your JS code. This directly triggers the debugger to start and allows you to inspect your plot properly. Just right click on your „Viewer“ window in RStudio and select „Inspect Element“.



As soon as you hover over the plot, this invokes the debugger and allows you to see the complete JS element with all data and attributes.



That’s basically the equivalent to calling rbokeh::debug\_callback if you didn’t use your own JS code.  
Another nice JS function to know is console.log(). Whatever element or variable you want to have a closer look at, just put it inside this function and check what’s behind it via „Inspect Element/Console“. Very helpful to see whether something looks like or contains what you expect it would. Here is an example with console.log(points\_data.get('data')).



**Is there more we can do to customize rBokeh..?**

This is all I got for you concerning callbacks in rBokeh. In my next blog post, I will show you how to manipulate a standard rBokeh plot as extensively as its counterpart in Python. Stay tuned!