

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"

) %>%

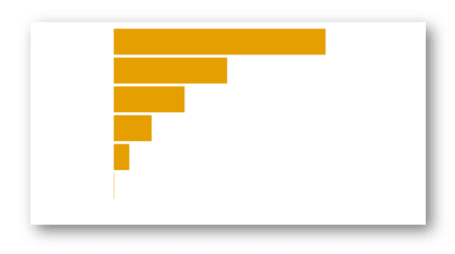
r2d3(r2d3\_file)

# ^^ 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;

Using r2d3 with Shiny

**Overview**

Integrating Shiny and D3 can be a powerful combination. Deriving insights from data is streamlined when users are able to modify a Shiny input, or click on a D3 visualization, and that action produces new results.

There are two ways that D3 visualizations can be integrated with Shiny. The difference has to do with the direction of the communication:

1. From **Shiny** to **D3**
2. From **D3** to **Shiny**

**Shiny to D3**

When going from Shiny to D3, users make changes to Shiny inputs which result in changes to the D3 visualization. The r2d3 package provides two functions that enables this functionality:

1. renderD3() - Works just like renderPlot(), it creates a reactive D3 plot that can be used as an output. renderD3() is used server-side (server):
2. server <- function(input, output) {
3. output$d3 <- renderD3({...})

}

1. d3Output() - Akin to plotOutput(), it is used client-side (ui):
2. ui <- fluidPage(
3. d3Output("d3")

)

Example

This example shows a Shiny app that modifies a D3 plot’s bar length based on the value that the user chooses for the sliderInput():

library(shiny)

library(r2d3)

ui <- fluidPage(

inputPanel(

sliderInput("bar\_max", label = "Max:",

min = 0, max = 1, value = 1, step = 0.05)

),

d3Output("d3")

)

server <- function(input, output) {

output$d3 <- renderD3({

r2d3(

runif(5, 0, input$bar\_max),

script = system.file("examples/baranims.js", package = "r2d3")

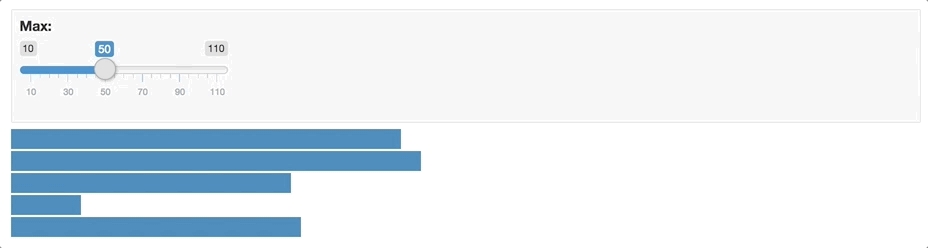
)

})

}

shinyApp(ui = ui, server = server)

And here is the app in action:



**D3 to Shiny**

Having a D3 plot update based on a changing Shiny input is very common. But, what about having an action that happened inside a D3 plot trigger something in Shiny?

For example, we may want Shiny to know that our D3 bar plot was clicked on, and know which bar was clicked. Enabling communication from D3 to Shiny requires changes to both D3 and Shiny code:

* *D3 script* - Add a call to the Shiny JavaScript function Shiny.setInputValue(). To use Shiny.setInputValue(), **you will need shiny version 1.1 or above.**
* *Shiny script* - Add a reactive function that looks for the change and performs an action based on the selected value.

Example

In this example, the bar plot will pass to Shiny the data value of the bar that the user clicked on. Shiny will then display the cosine of the value.

D3 Code

Here is the D3 script:

// !preview r2d3 data=c(0.3, 0.6, 0.8, 0.95, 0.40, 0.20)

var barHeight = Math.ceil(height / data.length);

svg.selectAll('rect')

.data(data)

.enter().append('rect')

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

.attr('height', barHeight)

.attr('y', function(d, i) { return i \* barHeight; })

.attr('fill', 'steelblue')

.attr("d", function(d) { return d; })

.on("click", function(){

Shiny.setInputValue(

"bar\_clicked",

d3.select(this).attr("d"),

{priority: "event"}

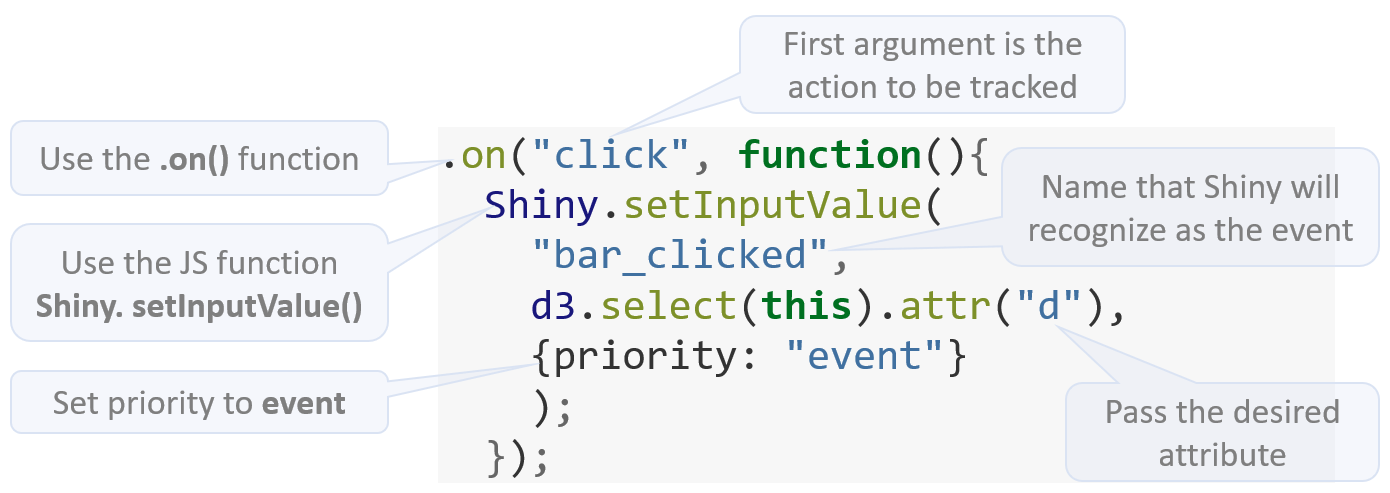
);

});

First, we need to decide which attribute (.attr) is going to be passed to Shiny when the user clicks. Any existing attribute can be passed, such as the value of y or width, but because no current attribute contain the raw value of the data, a new made up attribute is added, we called it d. The d attribute is set to rerun the d variable.



Next, the on() function is used to capture an event inside the plot.The first argument is *“click”*, indicating that we want it to be activated when someone clicks on the plot. The second argument of on() is a function, this function will contain a call to the Shiny.setInputValue() JS function. The first argument of the Shiny JS function is the name that the Shiny app will be looking for. In this case, we will use **bar\_clicked**. The second argument is the value that is returned to Shiny, and for that, we will use the d attribute created previously. The third argument, {priority: "event"}, is an option for input values that want to be treated less like values to be monitored, and more like events that are not to be missed.



Shiny Code

Here’s the code for the Shiny application that interacts with D3:

library(r2d3)

ui <- fluidPage(

verbatimTextOutput("selected"),

d3Output("d3")

)

server <- function(input, output) {

output$d3 <- renderD3({

r2d3(

c(0.3, 0.6, 0.8, 0.95, 0.40),

script = "bar.js"

)

})

output$selected <- renderText({

bar\_number <- as.numeric(req(input$bar\_clicked))

if (bar\_number > 0) cos(bar\_number)

})

}

shinyApp(ui, server)

The ui contains a text output named selected. Server-side, a reactive function, called output$selected() will used to capture the changes of the bar selection.

In the D3 script, we called Shiny.setInputValued() with an input named bar\_clicked. This can now be used as if was an Shiny input in the app. In this case, the name to track will be: input$bar\_clicked. **The Shiny.setInputValued() JS function is recognized by the shiny package version 1.1 and above.**

The output$selected() function uses renderText() to display the cosine of the bar that was clicked on. The call to input$bar\_clicked is used inside renderText() to execute the cos() calculation.

The default value of input$bar\_clicked will be NULL at the startup of the app. In this state you won’t want to perform any calculation (as it would result in an error) or display any output. Shiny’s req() function can be used to prevent errors in this case req() stops reactivity if the value of the variable passed as an argument is not valid.

Here is the Shiny application in action:



* 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. 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.
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 on db.rstudio.com;

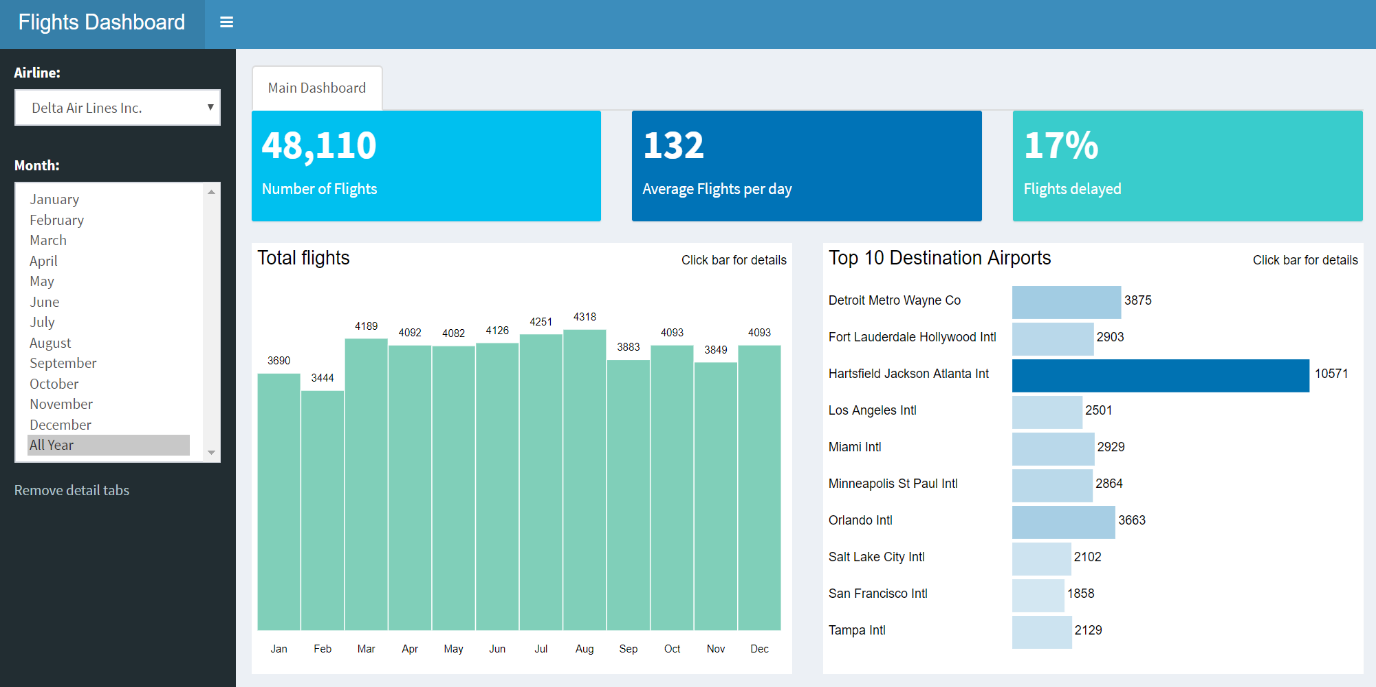
Enterprise Dashboard Shiny App Code

Design principles

A few principles to keep in mind when developing an enterprise level dashboard:

* Push as much of the calculations of the dashboard back to the database - The time it takes for a dashboard to load, and respond, will become the most important aspect of its design. For dashboards, the expected time to load and response is a few seconds.
* Give the end-user with “train of thought” paths - These paths are commonly provided by way of a drill down from within the dashboard. These paths allows the end-user to quickly answer questions they currently have.
* Make data driven inputs - It is easy to “hard code” the values that will be available in an input, such as a drop down. This may become a problem later on if the possible values change over time. An ancillary principle is to always use a look up table to obtain the list of values, when available. It is not ideal to obtain a list of possible values by grouping and summarizing a large column.
* Secure the database credentials - Most often, a service account is made available for reports and dashboards that have access to a database. In that case, it is important to know how to properly safeguard the credentials of the service account.

Example

[](https://edgarruiz.shinyapps.io/db-dashboard/)Click to see the live dashboard

A working example of the dashboard that will be used as the base of this article is available on GitHub. The project repository contains the code and all the supporting files:

* Visit the repository: https://github.com/sol-eng/db-dashboard
* Download a zipped file with all of the files from the repository: https://github.com/sol-eng/db-dashboard/archive/master.zip

A live version of the app can be found in shinyapps.io: DB Dashboard

Use shinydashboard

The shinydashboard package has three important advantages:

1. Provides an out-of-the-box framework to create dashboards in Shiny. This saves a lot of time, because the developer does not have to create the dashboard features manually using “base” Shiny.
2. Has a dashboard-firendly tag structure. This allows the developer to get started quickly. Inside the dashboardPage()tag, the dashboardHeader(), dashboardSidebar() and dashboardBody() can be added to easily lay out a new dashboard.
3. It is mobile-ready. Without any additional code, the dashboard layout will adapt to a smaller screen automatically.

Another option for creating dashboard with R is flexdashboard. It will support all but one of the features discussed in this article: dynamic tabs, which are the basis of the technique used in the example to drive the drill throughs.

QUICK EXAMPLE

If you are new to shinydashboard, please feel free to copy and paste the following code to see a very simple dashboard in your environment:

**library**(shinydashboard)

**library**(shiny)

ui <- dashboardPage(

dashboardHeader(title = "Quick Example"),

dashboardSidebar(textInput("text", "Text")),

dashboardBody(

valueBox(100, "Basic example"),

tableOutput("mtcars")

)

)

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

output$mtcars <- renderTable(head(mtcars))

}

shinyApp(ui, server)

Connection strings and credentials

In many cases, the app is developed against one database, and run in production against a different database. This can present a challenge if the connection string is “hard coded”.

This site provides a couple of articles to help with addressing credentials and portability of code:

* Securing Deployed Content
* Making Scripts Portable

Populate Shiny inputs using purrr

The usual preference is for the values displayed in a user input, such as drop down, to be “human readable”. The actual value of the selection should be a unique identifier so that dependent queries return the correct information.

This section examines two cases and ways to format the list of options to be in a format that Shiny can use.

DROP DOWN POPULATED FROM DATABASE

Ideally, a look up table is available in the database so that the query is simple to execute.

To separate the keys from the values, the map() function in the purrr package can be used. In the example below, all of the records in the airlines table are collected, and a list of names is created, map() is then used to insert the carrier codes into each name node.

airline\_list <- tbl(con, "airlines") %>%

collect %>%

split(.$name) %>% *# Field that will be used for the labels*

map(~.$carrier) *# Field that will be used for keys*

The selectInput() drop-down menu is able to read the resulting airline\_list list variable.

dashboardSidebar(

selectInput(

inputId = "airline",

label = "Airline:",

choices = airline\_list,

selected = "DL",

selectize = FALSE

)

LIST POPULATED FROM A VECTOR

There are times when the possible values are static, and small enough, so that they all fit in a vector.

A common example is the month name. A given table store the month number, and that number is what needs to be used as the filter value The values presented to the end-user will be the month name, but when a selection is made, the month number is what will be passed to Shiny.

For that, a function called set\_names() can be used to add the caption that will be displayed in the input in a way that is “Shiny friendly”

month\_list <- as.list(1:12) %>%

set\_names(month.name)

month\_list$`All Year` <- 99

The selectInput() list menu is able to read the resulting month\_list list variable.

selectInput(

inputId = "month",

label = "Month:",

choices = month\_list,

selected = 99,

size = 13,

selectize = FALSE

)

Create a base query using dplyr

In most cases, all of the plots and tables in a dashboard share a common base query. For example, they will all show the same month’s data. Using dplyr to build the base query has the following advantages:

* Simplifies the code because it prevents the repetition of filters and joins.
* dplyr “laziness” allows for the base query to be built with out it being executed until it is used to get the data for a given plot or table.
* Abstracts the translation of the SQL syntax. The dashboard will work with no, or minimal, changes if the database vendor changes.
* The modular nature of this approach allows to just add a few simple, and easy to understand, dplyr steps to get the slice or aggregation of the data needed to be displayed on the plot or table.

Because the base query will more likely have to be assembled based on the current input selection, then a Shiny reactive() function is necessary to be used instead of a regular function(). This is because the input$... variables can only be evaluated inside a Shiny reactive function.

base\_flights <- reactive({

res <- flights %>%

filter(carrier == input$airline) %>%

left\_join(airlines, by = "carrier") %>%

rename(airline = name) %>%

left\_join(airports, by = c("origin" = "faa")) %>%

rename(origin\_name = name) %>%

select(-lat, -lon, -alt, -tz, -dst) %>%

left\_join(airports, by = c("dest" = "faa")) %>%

rename(dest\_name = name)

**if** (input$month != 99) res <- filter(res, month == input$month)

res

})

Then, the Shiny output function starts with the base query (base\_flights), and finishing dplyr steps, in the form of verbs, are appended, and piped directly to the plotting or display function. It is important to note that before sending the resulting data set to Shiny, either a collect() or pull() function needs to be used.

output$per\_day <- renderValueBox({

base\_flights() %>% *#------ Base query*

group\_by(day, month) %>% *#-- Finishing steps*

tally() %>%

summarise(avg = mean(n)) %>%

pull() %>%

round() %>%

prettyNum(big.mark = ",") %>%

valueBox( *# -- Pipe right into a Value Box*

subtitle = "Average Flights per day",

color = "blue"

)

})

Using r2d3 for interactivity and drill-down

A “drill-down” is a great way to provide the end-user with “train of thought” paths.

In a Shiny app or dashboard, there’s the R object that contains the plot or table needs a way to pass to Shiny the value what was that was clicked on. The best way to do this, is by using Shiny’s JavaScript inside a given plot. This activates a reactive function inside the app.

The visualization packages called htmlwidgets are widely used. They are a set of packages are wrappers on top of D3/JavaScript plots. There may be times when the available htmlwidgets package falls short, either by not integrating with Shiny, or by not providing the exact visualization that is needed for the dashboard.

This article, the package r2d3 will be used. This package allows us to custom build D3 visualizations from the ground up, for maximum flexibility and best integration with Shiny. A more in-depth article on how to integrate Shiny with r2d3 is available here: Using r2d3 with Shiny.

TWO READY-TO-USE R2D3 PLOTS

The example dashboard used in this article contains two D3 scripts that are “Shiny-ready”. One is a column plot and the other a bar plot. They have been developed in a way that you can easily copy the entire script and use it in your own dashboard.

* col\_plot.js - Requires a data.frame or tibble with the following names and type of data:
  + x - Expects the category’s value. For example, if it represents a month, then it would contain the month’s number.
  + y - Expects the value of the height of the column.
  + label - Expects the category’s caption. It is what will be displayed to the end-user. For example, if it represents a month, then it would contain the month’s name.
* bar\_plot.js - - Requires a data.frame or tibble with the following names and type of data:
  + x - Expects the value of the width of the bar.
  + y - Expects the category’s value. For example, if it represents a month, then it would contain the month’s number.
  + label - Expects the category’s caption. It is what will be displayed to the end-user. For example, if it represents a month, then it would contain the month’s name.

Thanks to r2d3, the plots can easily be rendered. This code snippet shows how simple is to combine the technique of using a base query, and then pipe the finishing transformations directly into the r2d3() function.

output$top\_airports <- renderD3({

*# The following code runs inside the database*

base\_flights() %>%

group\_by(dest, dest\_name) %>%

tally() %>%

collect() %>%

arrange(desc(n)) %>%

head(10) %>%

arrange(dest\_name) %>%

mutate(dest\_name = str\_sub(dest\_name, 1, 30)) %>%

rename(

x = dest, *# Make sure to rename the*

y = n, *# variables to what the*

label = dest\_name *# D3 script expects*

) %>%

r2d3("bar\_plot.js")

})

Handling a click event from the plot

The ideal outcome of a click event is that it activates a Shiny input. This allows the app to execute a reactive function when the click, or any other event recognized by the plot, is triggered.

The D3 plots, available in the example’s GitHub repository, already contain the necessary Shiny JS code to trigger a reactive function when clicked on:

* col\_plot.js - Creates a input$col\_clicked inside the Shiny app.
* bar\_plot.js - Creates a input$bar\_clicked inside the Shiny app.

Inside the app, include an observeEvent() function that will capture the value returned by the D3 plot:

observeEvent(input$bar\_clicked, {

*# ----- Function's code --------*

})

Troubleshooting tip - If the nothing happens when a bar is clicked on, please confirm that the installed shiny package version is 1.1.0 or above.

Create the drill-down report

USING APPENDTAB() TO CREATE THE DRILL-DOWN REPORT

The plan is to display a new drill-down report every time the end user clicks on a bar. To prevent pulling the same data unnecessarily, the code will be “smart” enough to simply switch the focus to an existing tab if the same bar has been clicked on before. This switch also prevent unnecessary trips to the database.

The new, and really cool, appendTab() function is used to dynamically create a new Shiny tab with a data table from the DT package that contains the first 100 rows of the selection. A simple vector, called tab\_list, is used to track all existing detail tabs. The updateTabsetPanel() function is used to switch to the newly or previously created tab.

The observeEvent() function is the one that “catches” the event executed by the D3 code. It monitors the bar\_clicked Shiny input.

observeEvent(input$bar\_clicked, {

airport <- input$bar\_clicked

month <- input$month

tab\_title <- paste(

input$airline, "-", airport,

**if** (month != 99) {

paste("-", month.name[as.integer(month)])

}

)

**if** (!(tab\_title %**in**% tab\_list)) {

appendTab(

inputId = "tabs",

tabPanel(

tab\_title,

DT::renderDataTable(

*# This function return a data.frame with*

*# the top 100 records of that airport*

get\_details(airport = airport)

)

)

)

tab\_list <<- c(tab\_list, tab\_title)

}

updateTabsetPanel(session, "tabs", selected = tab\_title)

})

REMOVE ALL TABS USING REMOVETAB() AND PURRR

Creating new tabs dynamically can clutter the dashboard. So a simple actionLink() button can be added to the dashboardSidebar() in order to remove all tabs except the main dashboard tab.

*# This code runs in ui*

dashboardSidebar(

actionLink("remove", "Remove detail tabs"))

The observeEvent() function is used once more to catch when the link is clicked. The walk() command from purrr is then used to iterate through each tab title in the tab\_list vector and proceeds to execute the Shiny removeTab() command for each name. After that, the tab list variable is reset. Because of environment scoping, make sure to use double less than ( <<- ) when resetting the variable, so it knows to reset the variable defined outside of the observeEvent() function.

*# This code runs in server*

observeEvent(input$remove,{

*# Use purrr's walk command to cycle through each*

*# panel tabs and remove them*

tab\_list %>%

walk(~removeTab("tabs", .x))

tab\_list <<- NULL

})

1. Local\_App.R – Creating Shiny Application \_ Example to connect with Database Connection

|  |
| --- |
| library(shiny) |
|  | library(shinydashboard) |
|  |  |
|  | library(dplyr) |
|  | library(purrr) |
|  | library(rlang) |
|  | library(stringr) |
|  |  |
|  | library(DT) |
|  | library(r2d3) |
|  |  |
|  | library(nycflights13) |
|  |  |
|  | # Use purrr's split() and map() function to create the list |
|  | # needed to display the name of the airline but pass its |
|  | # Carrier code as the value |
|  |  |
|  | airline\_list <- airlines %>% |
|  | collect() %>% |
|  | split(.$name) %>% |
|  | map(~ .$carrier) |
|  |  |
|  | # Use rlang's set\_names() to easily create a valide "choices" |
|  | # argument of the dropdown where the displayed text has to be |
|  | # different than the value passed as the input selection |
|  |  |
|  | month\_list <- as.list(1:12) %>% |
|  | set\_names(month.name) |
|  |  |
|  | month\_list$`All Year` <- 99 |
|  |  |
|  | ui <- dashboardPage( |
|  | dashboardHeader( |
|  | title = "Flights Dashboard", |
|  | titleWidth = 200 |
|  | ), |
|  | dashboardSidebar( |
|  | selectInput( |
|  | inputId = "airline", |
|  | label = "Airline:", |
|  | choices = airline\_list, |
|  | selected = "DL", |
|  | selectize = FALSE |
|  | ), |
|  | sidebarMenu( |
|  | selectInput( |
|  | inputId = "month", |
|  | label = "Month:", |
|  | choices = month\_list, |
|  | selected = 99, |
|  | size = 13, |
|  | selectize = FALSE |
|  | ), |
|  | actionLink("remove", "Remove detail tabs") |
|  | ) |
|  | ), |
|  | dashboardBody( |
|  | tabsetPanel( |
|  | id = "tabs", |
|  | tabPanel( |
|  | title = "Main Dashboard", |
|  | value = "page1", |
|  | fluidRow( |
|  | valueBoxOutput("total\_flights"), |
|  | valueBoxOutput("per\_day"), |
|  | valueBoxOutput("percent\_delayed") |
|  | ), |
|  | fluidRow(), |
|  | fluidRow( |
|  | column( |
|  | width = 6, |
|  | d3Output("group\_totals") |
|  | ), |
|  | column( |
|  | width = 6, |
|  | d3Output("top\_airports") |
|  | ) |
|  | ) |
|  | ) |
|  | ) |
|  | ) |
|  | ) |
|  |  |
|  | server <- function(input, output, session) { |
|  | tab\_list <- NULL |
|  |  |
|  | # Use a reactive() function to prepare the base |
|  | # SQL query that all the elements in the dashboard |
|  | # will use. The reactive() allows us to evaluate |
|  | # the input variables |
|  | base\_flights <- reactive({ |
|  | res <- flights %>% |
|  | filter(carrier == input$airline) %>% |
|  | left\_join(airlines, by = "carrier") %>% |
|  | rename(airline = name) %>% |
|  | left\_join(airports, by = c("origin" = "faa")) %>% |
|  | rename(origin\_name = name) %>% |
|  | select(-lat, -lon, -alt, -tz, -dst) %>% |
|  | left\_join(airports, by = c("dest" = "faa")) %>% |
|  | rename(dest\_name = name) |
|  | if (input$month != 99) res <- filter(res, month == input$month) |
|  | res |
|  | }) |
|  |  |
|  | # Total Flights (server) ------------------------------------------ |
|  | output$total\_flights <- renderValueBox({ |
|  | # The following code runs inside the database. |
|  | # pull() bring the results into R, which then |
|  | # it's piped directly to a valueBox() |
|  | base\_flights() %>% |
|  | tally() %>% |
|  | pull() %>% |
|  | as.integer() %>% |
|  | prettyNum(big.mark = ",") %>% |
|  | valueBox(subtitle = "Number of Flights") |
|  | }) |
|  |  |
|  | # Avg per Day (server) -------------------------------------------- |
|  | output$per\_day <- renderValueBox({ |
|  | # The following code runs inside the database |
|  | base\_flights() %>% |
|  | group\_by(day, month) %>% |
|  | tally() %>% |
|  | ungroup() %>% |
|  | summarise(avg = mean(n)) %>% |
|  | pull(avg) %>% |
|  | round() %>% |
|  | prettyNum(big.mark = ",") %>% |
|  | valueBox( |
|  | subtitle = "Average Flights per day", |
|  | color = "blue" |
|  | ) |
|  | }) |
|  |  |
|  | # Percent delayed (server) ---------------------------------------- |
|  | output$percent\_delayed <- renderValueBox({ |
|  | base\_flights() %>% |
|  | filter(!is.na(dep\_delay)) %>% |
|  | mutate(delayed = ifelse(dep\_delay >= 15, 1, 0)) %>% |
|  | summarise( |
|  | delays = sum(delayed), |
|  | total = n() |
|  | ) %>% |
|  | mutate(percent = (delays / total) \* 100) %>% |
|  | pull() %>% |
|  | round() %>% |
|  | paste0("%") %>% |
|  | valueBox( |
|  | subtitle = "Flights delayed", |
|  | color = "teal" |
|  | ) |
|  | }) |
|  |  |
|  | # Montly/daily trend (server) ------------------------------------- |
|  | output$group\_totals <- renderD3({ |
|  | grouped <- ifelse(input$month != 99, expr(day), expr(month)) |
|  |  |
|  | res <- base\_flights() %>% |
|  | group\_by(!!grouped) %>% |
|  | tally() %>% |
|  | collect() %>% |
|  | mutate( |
|  | y = n, |
|  | x = !!grouped |
|  | ) %>% |
|  | select(x, y) |
|  |  |
|  | if (input$month == 99) { |
|  | res <- res %>% |
|  | inner\_join( |
|  | tibble(x = 1:12, label = substr(month.name, 1, 3)), |
|  | by = "x" |
|  | ) |
|  | } else { |
|  | res <- res %>% |
|  | mutate(label = x) |
|  | } |
|  | r2d3(res, "col\_plot.js") |
|  | }) |
|  |  |
|  | # Top airports (server) ------------------------------------------- |
|  | output$top\_airports <- renderD3({ |
|  | # The following code runs inside the database |
|  | base\_flights() %>% |
|  | group\_by(dest, dest\_name) %>% |
|  | tally() %>% |
|  | collect() %>% |
|  | arrange(desc(n)) %>% |
|  | head(10) %>% |
|  | arrange(dest\_name) %>% |
|  | mutate(dest\_name = str\_sub(dest\_name, 1, 30)) %>% |
|  | rename( |
|  | x = dest, |
|  | y = n, |
|  | label = dest\_name |
|  | ) %>% |
|  | r2d3("bar\_plot.js") |
|  | }) |
|  |  |
|  | # Get details (server) -------------------------------------------- |
|  | get\_details <- function(airport = NULL, day = NULL) { |
|  | # Create a generic details function that can be called |
|  | # by different dashboard events |
|  | res <- base\_flights() |
|  | if (!is.null(airport)) res <- filter(res, dest == airport) |
|  | if (!is.null(day)) res <- filter(res, day == !!as.integer(day)) |
|  |  |
|  | res %>% |
|  | head(100) %>% |
|  | select( |
|  | month, day, flight, tailnum, |
|  | dep\_time, arr\_time, dest\_name, |
|  | distance |
|  | ) %>% |
|  | collect() %>% |
|  | mutate(month = month.name[as.integer(month)]) |
|  | } |
|  |  |
|  | # Month/Day column click (server) --------------------------------- |
|  | observeEvent(input$column\_clicked != "", { |
|  | if (input$month == "99") { |
|  | updateSelectInput(session, "month", selected = input$column\_clicked) |
|  | } else { |
|  | day <- input$column\_clicked |
|  | month <- input$month |
|  | tab\_title <- paste( |
|  | input$airline, "-", month.name[as.integer(month)], "-", day |
|  | ) |
|  | if (!(tab\_title %in% tab\_list)) { |
|  | appendTab( |
|  | inputId = "tabs", |
|  | tabPanel( |
|  | tab\_title, |
|  | DT::renderDataTable( |
|  | get\_details(day = day) |
|  | ) |
|  | ) |
|  | ) |
|  | tab\_list <<- c(tab\_list, tab\_title) |
|  | } |
|  | updateTabsetPanel(session, "tabs", selected = tab\_title) |
|  | } |
|  | }, |
|  | ignoreInit = TRUE |
|  | ) |
|  |  |
|  |  |
|  | # Bar clicked (server) -------------------------------------------- |
|  | observeEvent(input$bar\_clicked, { |
|  | airport <- input$bar\_clicked |
|  | month <- input$month |
|  | tab\_title <- paste( |
|  | input$airline, "-", airport, |
|  | if (month != 99) { |
|  | paste("-", month.name[as.integer(month)]) |
|  | } |
|  | ) |
|  | if (!(tab\_title %in% tab\_list)) { |
|  | appendTab( |
|  | inputId = "tabs", |
|  | tabPanel( |
|  | tab\_title, |
|  | DT::renderDataTable( |
|  | get\_details(airport = airport) |
|  | ) |
|  | ) |
|  | ) |
|  |  |
|  | tab\_list <<- c(tab\_list, tab\_title) |
|  | } |
|  | updateTabsetPanel(session, "tabs", selected = tab\_title) |
|  | }) |
|  |  |
|  | # Remote tabs (server) -------------------------------------------- |
|  | observeEvent(input$remove, { |
|  | # Use purrr's walk command to cycle through each |
|  | # panel tabs and remove them |
|  | tab\_list %>% |
|  | walk(~ removeTab("tabs", .x)) |
|  | tab\_list <<- NULL |
|  | }) |
|  | } |
|  |  |
|  | shinyApp(ui, server) |

Db\_app.R- It shows how to connect with database from Shiny App

|  |
| --- |
| library(shiny) |
|  | library(shinydashboard) |
|  |  |
|  | library(dplyr) |
|  | library(purrr) |
|  | library(rlang) |
|  | library(stringr) |
|  |  |
|  | library(DT) |
|  | library(r2d3) |
|  |  |
|  | library(odbc) |
|  | library(DBI) |
|  | library(dbplyr) |
|  | library(config) |
|  |  |
|  | dw <- config::get("mssql") |
|  |  |
|  | con <- DBI::dbConnect( |
|  | odbc::odbc(), |
|  | DSN = dw$DSN |
|  | ) |
|  |  |
|  | airlines <- tbl(con, "airlines") |
|  | airports <- tbl(con, "airports") |
|  | flights <- tbl(con, "flights") |
|  |  |
|  | # Use purrr's split() and map() function to create the list |
|  | # needed to display the name of the airline but pass its |
|  | # Carrier code as the value |
|  |  |
|  | airline\_list <- airlines %>% |
|  | collect() %>% |
|  | split(.$name) %>% |
|  | map(~ .$carrier) |
|  |  |
|  | # Use rlang's set\_names() to easily create a valide "choices" |
|  | # argument of the dropdown where the displayed text has to be |
|  | # different than the value passed as the input selection |
|  |  |
|  | month\_list <- as.list(1:12) %>% |
|  | set\_names(month.name) |
|  |  |
|  | month\_list$`All Year` <- 99 |
|  |  |
|  | ui <- dashboardPage( |
|  | dashboardHeader( |
|  | title = "Flights Dashboard", |
|  | titleWidth = 200 |
|  | ), |
|  | dashboardSidebar( |
|  | selectInput( |
|  | inputId = "airline", |
|  | label = "Airline:", |
|  | choices = airline\_list, |
|  | selected = "DL", |
|  | selectize = FALSE |
|  | ), |
|  | sidebarMenu( |
|  | selectInput( |
|  | inputId = "month", |
|  | label = "Month:", |
|  | choices = month\_list, |
|  | selected = 99, |
|  | size = 13, |
|  | selectize = FALSE |
|  | ), |
|  | actionLink("remove", "Remove detail tabs") |
|  | ) |
|  | ), |
|  | dashboardBody( |
|  | tabsetPanel( |
|  | id = "tabs", |
|  | tabPanel( |
|  | title = "Main Dashboard", |
|  | value = "page1", |
|  | fluidRow( |
|  | valueBoxOutput("total\_flights"), |
|  | valueBoxOutput("per\_day"), |
|  | valueBoxOutput("percent\_delayed") |
|  | ), |
|  | fluidRow(), |
|  | fluidRow( |
|  | column( |
|  | width = 6, |
|  | d3Output("group\_totals") |
|  | ), |
|  | column( |
|  | width = 6, |
|  | d3Output("top\_airports") |
|  | ) |
|  | ) |
|  | ) |
|  | ) |
|  | ) |
|  | ) |
|  |  |
|  | server <- function(input, output, session) { |
|  | tab\_list <- NULL |
|  |  |
|  | # Use a reactive() function to prepare the base |
|  | # SQL query that all the elements in the dashboard |
|  | # will use. The reactive() allows us to evaluate |
|  | # the input variables |
|  | base\_flights <- reactive({ |
|  | res <- flights %>% |
|  | filter(carrier == input$airline) %>% |
|  | left\_join(airlines, by = "carrier") %>% |
|  | rename(airline = name) %>% |
|  | left\_join(airports, by = c("origin" = "faa")) %>% |
|  | rename(origin\_name = name) %>% |
|  | select(-lat, -lon, -alt, -tz, -dst) %>% |
|  | left\_join(airports, by = c("dest" = "faa")) %>% |
|  | rename(dest\_name = name) |
|  | if (input$month != 99) res <- filter(res, month == input$month) |
|  | res |
|  | }) |
|  |  |
|  | # Total Flights (server) ------------------------------------------ |
|  | output$total\_flights <- renderValueBox({ |
|  | # The following code runs inside the database. |
|  | # pull() bring the results into R, which then |
|  | # it's piped directly to a valueBox() |
|  | base\_flights() %>% |
|  | tally() %>% |
|  | pull() %>% |
|  | as.integer() %>% |
|  | prettyNum(big.mark = ",") %>% |
|  | valueBox(subtitle = "Number of Flights") |
|  | }) |
|  |  |
|  | # Avg per Day (server) -------------------------------------------- |
|  | output$per\_day <- renderValueBox({ |
|  | # The following code runs inside the database |
|  | base\_flights() %>% |
|  | group\_by(day, month) %>% |
|  | tally() %>% |
|  | ungroup() %>% |
|  | summarise(avg = mean(n, na.rm = TRUE)) %>% |
|  | pull() %>% |
|  | round() %>% |
|  | prettyNum(big.mark = ",") %>% |
|  | valueBox( |
|  | subtitle = "Average Flights per day", |
|  | color = "blue" |
|  | ) |
|  | }) |
|  |  |
|  | # Percent delayed (server) ---------------------------------------- |
|  | output$percent\_delayed <- renderValueBox({ |
|  | base\_flights() %>% |
|  | filter(!is.na(dep\_delay)) %>% |
|  | mutate(delayed = ifelse(dep\_delay >= 15, 1, 0)) %>% |
|  | summarise( |
|  | delays = sum(delayed, na.rm = TRUE), |
|  | total = n() |
|  | ) %>% |
|  | mutate(percent = (delays / total) \* 100) %>% |
|  | pull() %>% |
|  | round() %>% |
|  | paste0("%") %>% |
|  | valueBox( |
|  | subtitle = "Flights delayed", |
|  | color = "teal" |
|  | ) |
|  | }) |
|  |  |
|  | # Montly/daily trend (server) ------------------------------------- |
|  | output$group\_totals <- renderD3({ |
|  | grouped <- ifelse(input$month != 99, expr(day), expr(month)) |
|  |  |
|  | res <- base\_flights() %>% |
|  | group\_by(!!grouped) %>% |
|  | tally() %>% |
|  | collect() %>% |
|  | mutate( |
|  | y = n, |
|  | x = !!grouped |
|  | ) %>% |
|  | select(x, y) |
|  |  |
|  | if (input$month == 99) { |
|  | res <- res %>% |
|  | inner\_join( |
|  | tibble(x = 1:12, label = substr(month.name, 1, 3)), |
|  | by = "x" |
|  | ) |
|  | } else { |
|  | res <- res %>% |
|  | mutate(label = x) |
|  | } |
|  | r2d3(res, "col\_plot.js") |
|  | }) |
|  |  |
|  | # Top airports (server) ------------------------------------------- |
|  | output$top\_airports <- renderD3({ |
|  | # The following code runs inside the database |
|  | base\_flights() %>% |
|  | group\_by(dest, dest\_name) %>% |
|  | tally() %>% |
|  | collect() %>% |
|  | arrange(desc(n)) %>% |
|  | head(10) %>% |
|  | arrange(dest\_name) %>% |
|  | mutate(dest\_name = str\_sub(dest\_name, 1, 30)) %>% |
|  | rename( |
|  | x = dest, |
|  | y = n, |
|  | label = dest\_name |
|  | ) %>% |
|  | r2d3("bar\_plot.js") |
|  | }) |
|  |  |
|  | # Get details (server) -------------------------------------------- |
|  | get\_details <- function(airport = NULL, day = NULL) { |
|  | # Create a generic details function that can be called |
|  | # by different dashboard events |
|  | res <- base\_flights() |
|  | if (!is.null(airport)) res <- filter(res, dest == airport) |
|  | if (!is.null(day)) res <- filter(res, day == !!as.integer(day)) |
|  |  |
|  | res %>% |
|  | head(100) %>% |
|  | select( |
|  | month, day, flight, tailnum, |
|  | dep\_time, arr\_time, dest\_name, |
|  | distance |
|  | ) %>% |
|  | collect() %>% |
|  | mutate(month = month.name[as.integer(month)]) |
|  | } |
|  |  |
|  | # Month/Day column click (server) --------------------------------- |
|  | observeEvent(input$column\_clicked != "", { |
|  | if (input$month == "99") { |
|  | updateSelectInput(session, "month", selected = input$column\_clicked) |
|  | } else { |
|  | day <- input$column\_clicked |
|  | month <- input$month |
|  | tab\_title <- paste( |
|  | input$airline, "-", month.name[as.integer(month)], "-", day |
|  | ) |
|  | if (!(tab\_title %in% tab\_list)) { |
|  | appendTab( |
|  | inputId = "tabs", |
|  | tabPanel( |
|  | tab\_title, |
|  | DT::renderDataTable( |
|  | get\_details(day = day) |
|  | ) |
|  | ) |
|  | ) |
|  | tab\_list <<- c(tab\_list, tab\_title) |
|  | } |
|  | updateTabsetPanel(session, "tabs", selected = tab\_title) |
|  | } |
|  | }, |
|  | ignoreInit = TRUE |
|  | ) |
|  |  |
|  |  |
|  | # Bar clicked (server) -------------------------------------------- |
|  | observeEvent(input$bar\_clicked, { |
|  | airport <- input$bar\_clicked |
|  | month <- input$month |
|  | tab\_title <- paste( |
|  | input$airline, "-", airport, |
|  | if (month != 99) { |
|  | paste("-", month.name[as.integer(month)]) |
|  | } |
|  | ) |
|  | if (!(tab\_title %in% tab\_list)) { |
|  | appendTab( |
|  | inputId = "tabs", |
|  | tabPanel( |
|  | tab\_title, |
|  | DT::renderDataTable( |
|  | get\_details(airport = airport) |
|  | ) |
|  | ) |
|  | ) |
|  |  |
|  | tab\_list <<- c(tab\_list, tab\_title) |
|  | } |
|  | updateTabsetPanel(session, "tabs", selected = tab\_title) |
|  | }) |
|  |  |
|  | # Remote tabs (server) -------------------------------------------- |
|  | observeEvent(input$remove, { |
|  | # Use purrr's walk command to cycle through each |
|  | # panel tabs and remove them |
|  | tab\_list %>% |
|  | walk(~ removeTab("tabs", .x)) |
|  | tab\_list <<- NULL |
|  | }) |
|  | } |
|  |  |
|  | shinyApp(ui, server) |

**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.

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.