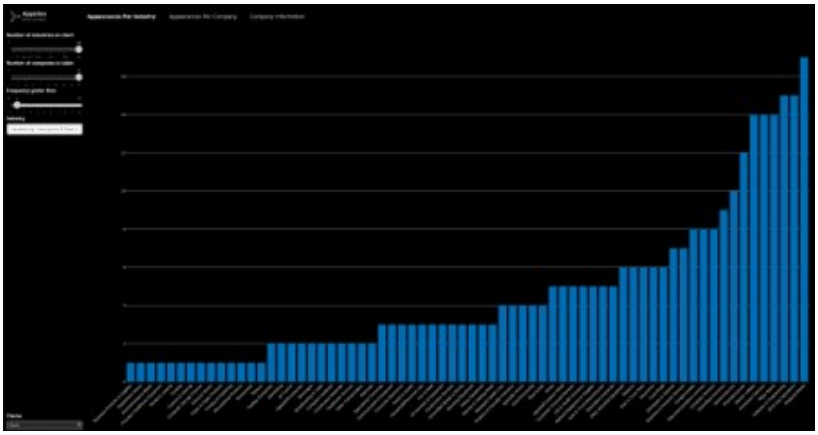
The opportunity to gain some skills in R Shiny came very quickly. I needed to prepare a report for my manager and an interactive dashboard seemed like the best way to present the huge amount of data I had. All of the developers were really busy in the projects and I needed to work on my own.

The only programming language I have ever touched is Python and I would say that I know it on a “google it and try it” level. I thought learning R would be quite a challenge. Fortunately I found out there are many examples of R Shiny code out there which I could use, so I started immediately. I was really surprised how easy it is to start writing code in R.

After two days I had a functional app which showed all of the data I wanted to display. It was easy to analyze the data which could be useful for business decisions. I created something that I knew would be used in the future.

The next day I felt like a pro and I wanted to improve my dashboard even more. And then the problems started. I realized that starting programming in shiny is easy but if you want to customize the app, it gets tricky. I was talking with Pedro, one of my colleagues and he offered help. I thought it was a good opportunity to learn from somebody more experienced. The results definitely exceeded my expectations.



To sum up, in my opinion R Shiny is a great way to quickly display data and create interactive dashboards. But when you want to have a custom design and you are not familiar with css and javascript it might be a bit harder. Another big issue is moving the app to production. My app is used by no more than 2 users concurrently, so It works fine. But If I would like it to be available to a few hundred people, it would require some extra steps. So don’t be afraid to create a Proof of Concept in R. Moving it to production is doable. In one of the projects I lead, we created an R Shiny app which is being used by 500 users.

Code

Let’s say that we have data that represents a set of industries, companies and the frequency of a certain event for each company:

Data:

match_name_table

Name
freq
Industry

industries_table

industry
freq

ui/server

First I learned about the structure of an R Shiny app. I found out that the main file is divided into two parts, ui and server. The ui is build from sidebarPanel and mainPanel. The sidebarPanel contains input controls which can be passed to the mainPanel. The mainPanel contains items defined on a server side which we would like to display in the app. As you can see below, my app contains four inputs, two plots, one table and a text field. I don't want to show all of the code at once because it is quite long. I used (…) in the places that will be explained later.

```
  library(jsonlite)
  library(dplyr)
  library(data.table)
  library(shiny)
  library(shinyWidgets)
  library(DT)
  library(plotly)

ui <- fluidPage(

  sidebarLayout(
    sidebarPanel(

      sliderInput(...),
      sliderInput(...),
      pickerInput(...),
      sliderInput(...),
      width = 2
    ),

    mainPanel(
      plotlyOutput("distPlot"),
      dataTableOutput("view"),
      tableOutput("view2"),
      plotlyOutput("distPlot2"),
      width = 10
    )
  ))

  server <- function(input, output, session) {
    no_of_companies <- reactiveVal(0, "Number of Companies:")

    observeEvent(..)
```

```
    observeEvent(...)
    output$distPlot <- renderPlotly(...)
    output$view <- DT::renderDataTable(...)
    output$distPlot2 <- renderPlotly(...)
    output$view2 <- renderText(...)
  }
  shinyApp(ui = ui, server = server)
```

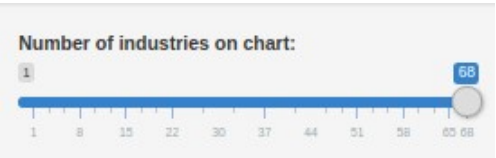
sliders

First I would like to write about the sliders. The data set used in the app was quite big and I wanted to give the user the possibility to filter it out. The first slider filters on a Freq value from the industries_table.

```
sliderInput(
    inputId = "freq",
    label = "Frequency grater then",
    min = 0,
    max = min(10 , max(as.numeric(industries_table["Freq"][, 1]))),
    value = 1
  )
```

Another thing was the number of bars on a first bar chart. I realized that the user might want to focus only on the Industries which are the most common in the data set. That is why I created the slider which allows the user to select the number of bars shown on a plot. This way the user could see only top n industries from the data set.

```
      sliderInput(
         inputId = "no_of_industries",
         label = "Number of industries on chart:",
         min = 1,
         max = length(industries_table["Freq"][, 1]),
         value = 5
       ),
```



The maximum value of the second slider is defined based on a number of rows in the industries_table table, which depends on the first slider input. I needed to update the maximal value for the second slider every time after user changed the value in the first slider. Here I used the observeEvent which allowed me to monitor the changes and update the slider input if necessary. The grayed out part is responsible for preparing the data and will be explained later. The code below tells us that we will monitor the freq slider and change the maximum and default value of no_of_industries slider.

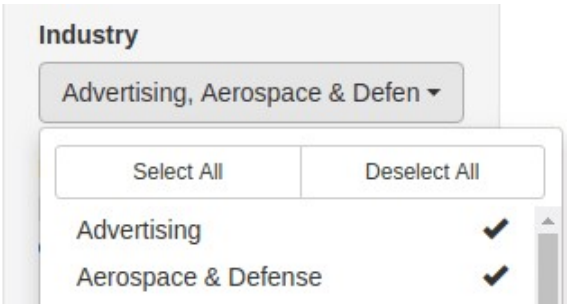
Then, I added one more slider and observer built in the same way as the one above.

```
observeEvent(input$freq, {
      min_value = input$freq + 1
      max_value = max(as.numeric(match_name_table["Freq"][,
1])
  match_name_table = match_name_table[match_name_table$Freq
%in% min_value:max_value), ]
```

```
      x   <- order_company_per_industry_
counts(match_name_table)
      updateSliderInput(
        session,
        inputId = "no_of_industries",
        min = 1,
        max = length(x),
        value = length(x)
      )
    })
```

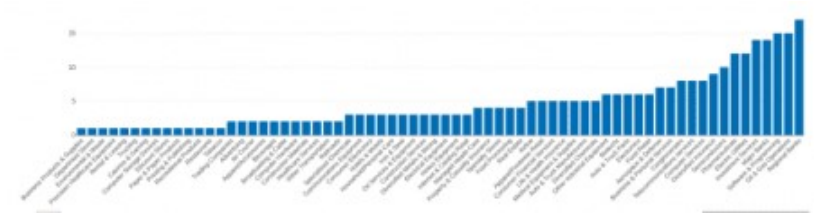
I also added a picker which enables the user to see only companies from the industries in which she is interested.

```
shinyWidgets::pickerInput(
        InputId = "industries",
        label = "Industry",
        choices = unique(match_name_table[3][
order(match_name_table[3]$industry), ]),
        selected = unique(match_name_table[3][
order(match_name_table[3]$industry), ]),
        options = list('actions-box' = TRUE),
        multiple = T
      ),
```



Plots

The first plot I created was a bar chart which showed the frequency for each industry. I used an R Shiny library called Plotly.



I assigned the renderPlotly function to the first output called distPlot.

```
    output$distPlot <- renderPlotly({
      min_value = input$freq + 1
      max_value = max(as.numeric(match_name_table["Freq"][,
1])
  match_name_table = match_name_table[match_name_table$Freq
%in% min_value:max_value),
      freq_values    <- order_company_per_industry_
counts(match_name_table)["Freq"][, 1]
      industries    <- order_company_per_industry_
counts(match_name_table)["industry"][, 1]
      n    <- as.integer(input$no_of_industries)
      freq_values    <- freq_values[(length(freq_values) - n
```

```
+ 1):length(freq_values)]
      industries    <- as.character(industries[(
length(industries) - n + 1):length(industries)])

      xform <- list(categoryorder = "array", categoryarray =
industries, tickangle = -45)
      plot_ly(y = freq_values, x = industries, type = "bar")
%>%
        layout(xaxis = xform)
    }
```

In the first part of the code you can see preparation of the data. The idea is to display the data only for companies that have a frequency bigger then min_value. The rows with frequency values lower than min_value are filtered out from match_name_table table. The min_value is defined based on the freq input.

Then the data used in a plot is prepared – freq_values and industries.

The industries names are assigned to industries and the number of companies which visited the web page is assigned to freq_values. The last thing we have to do is to limit the number of bars displayed on the plot to input$no_of_industries elements. In this way we have everything to build our bar chart.

A list called xform is used here to maintain the order of the industries and set the angle of the labels to 45 degrees. The default is to display the data alphabetically.

```
plot_ly(y = freq_values, x = industries, type = "bar") %>%
layout(xaxis = xform)
```

Is equivalent to

```
layout(plot_ly(y = values, x = industries, type = "bar"),
xaxis = xform)
```

order_company_per_industry_counts is a simple function which counts the number of companies for each industry and orders the data in descending order.

The order_company_per_industry_counts method looks like this:

```
order_company_per_industry_counts <-
function(match_name_table) {
  new_table = table(match_name_table$industry)
  new_table <- as.data.frame(t(new_table))
  new_table <- subset(new_table, select = -c(Var1))
  new_table <- new_table[order(new_table$Freq),]
  return(new_table)
}
```

reactive value

Then I added one more bar chart displaying the frequency for all companies. I thought it would also be nice to display the count of all of the companies from the plot. To achieve this I needed to use a reactive value. The value was updated in the plot body and displayed as a text.

```
no_of_companies <- reactiveVal(0, "Number of Companies:")
      output$distPlot2 <- renderPlotly({
      ...
      no_of_companies(nrow(data_plot))
      …
      }
    })
```

```
    output$view2 <- renderText({
      paste("Total number of companies meeting the
criteria:", no_of_companies())
    })
```

table

I thought that the user may want to see more detailed information about the companies. Here I decided to use the DT library.

```
output$view <- DT::renderDataTable({
      req(input$industries)
      min_value = input$freq + 1
     max_value = max(as.numeric(match_name_table["Freq"][,
1])
 match_name_table = match_name_table[match_name_table$Freq
%in% min_value:max_value), ]
      comp_data = match_name_table[match_name_table$industry
%in% input$industries,]
      data = merge(x = comp_data,
                           y = select(data_industries,
2,3,8,9,10,11,12),
                           by = c("name"),
                           all.x = TRUE)
      data <- data[order(data$Freq), ]
      if (nrow(data) == 0) {
        print("Please change the Frequency to a smaller
value")
      }
      else{
        tail(data, as.integer(input$no_of_
industries_in_a_table))
      }
    })
```

The interesting part here is the first line. Before I added it the app was displaying an error every time the industry's input was empty. Function req ensures that values are available before proceeding with an action.