Untitled

Peter Thramkrongart & Jakub Raszka

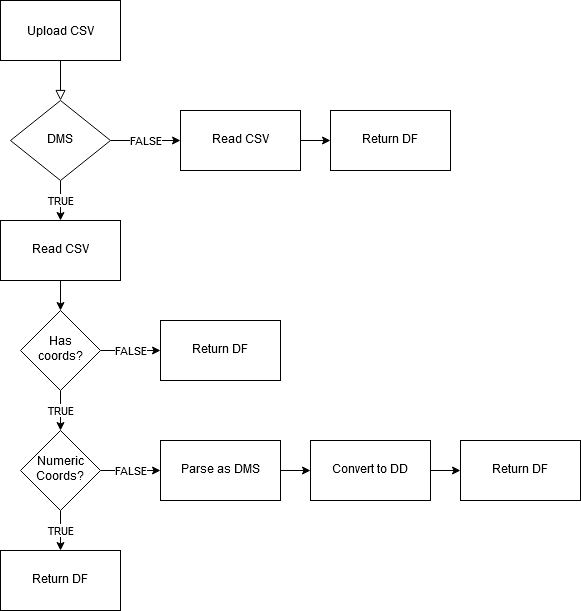
3/6/2021

## 5 Implementation

Our project is based on two core libraries: *shiny* and *leaflet*. Both are tools that are fairly easy to implement for R-novices and both provide powerful and robust functionality. Shiny is an R-based tool for building interactive web-applications. Its main strength is that it is controlled by the R Programming language. This means that all our code can be written in a couple of R-scripts. Similarly, *Leaflet* allows us to create interactive maps with a few lines of R code. In both cases, interactivity is key: Users need an interactive web-app to intuitively be able to create informative and good maps without too much expertise. In terms of novice map making, we find interactivity that interactivity enhances the user experience dramatically. Using the *web mercator* projection allows user to plot point data from all over the world at the same time, and the interactivity allows multiple levels of granularity and detail. For a travel map, user may plot multiple points in the cities they visits, and the may plot cities on multiple continents. We, therefore, find that leaflet provides the most functionality relevant to novices in a way that is very easy to use. If needed, novice R-user will easily be able to adopt the code we used for mapping for other projects. If, were to make static maps instead, *ggplot2* + *ggmap* (Kahle and Wickham 2013; Wickham 2016) and *tmap* (Tennekes 2018) would have been great candidates. Both have use easy to learn syntax, and both are very powerful. Tmap actually, uses Leaflet for its interactive mapping functionality. So does, the beginner-friendly *mapview* package(Appelhans et al. 2020). We, therefore, chose go to closer to the source and use the leaflet R-package.

Shiny apps consists of two main elements. A user interface (UI) script and and a server script. The scripts can be combined to one, but it will still consist of the two elements. The UI.R script is very similar to a markup script. It essentially, controls the layout and design of the app. The UI script is were we make and place the controls that the user can interact with such as checkboxes and upload/download buttons. The server.R script contains all th logic for the creation of the map, the handling of the downloaded data, the popup messages, the creation of the table and so on. The server has reactive elements that dynamically change as the user interact with controls of the UI. E.g. the user can choose different seperator values to parse the dataset, and the reactive data variable changes accordingly. Similarly, the server has observing objects such the monitoring error message popups that waits for the user to upload data or trigger the diagnosis button.

Because the app has many reactive objects, such as the map that changes depending on the dataset, we had to implement several measures to handle errors and avoid the app crashing. This turned out be a much bigger feat than we anticipated, and it will be a general theme throughout this project. The main problems arise the second the user uploads the tabulated data. If something goes wrong in the uploaded, the effects will trickle down to the reactive objects relying on the data format being correct. The expects two columns named *lat* and *long* to be the coordinates in DD. If columns does not exist, we still want to return the dataframe so that the user can access the data via the table in the app to debug the problem. We offer the option of converting the coordinates provided in the DMS-format to DD. If the user checks the DMS-conversion checkbox, the app will attempt to convert the coordinates to DD. But two conditions need to be met. First, it verifies presence of the the *lat* and long columns and, secondly, it checks whether they are in numeric format. Otherwise, the conversion would fail. We have created a flow chart that describes the main logic of the data upload.



Flowchart of the data upload logic

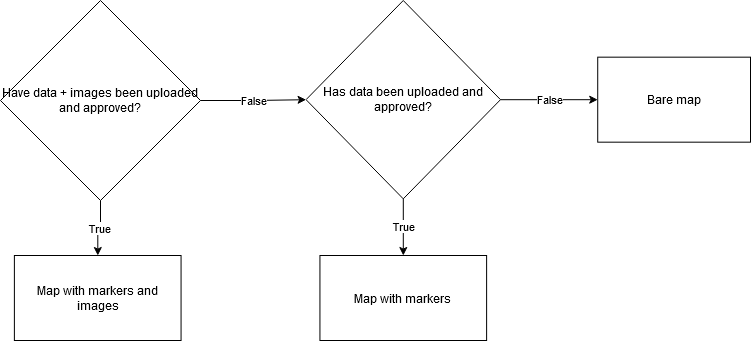
When the data is uploaded an observing error checker object is triggered. It checks that the provided data is in the right format, and attempts to diagnose some of the most commen errors, should they arise. It first checks that it has the coordinate columns. If they do not exist, it assumes that the data has been parsed with the wrong seperator. Next it ensures that all coordinates are numeric. Then it checks that the latitude and longitude within the proper ranges. Lastly, it ensures that a column with the name provided for the label column exist. If all is approved, the user will be given an popup message that informs the user that data has been succesfully uploaded and approved. Otherwise the error checker will provide the user with an appropriate error popup of the error highest in the logic tree. The code for the error functions has been attached below.

error\_check <- function(df, label\_col) {  
 len = length(df[, 1])  
   
 if (!sum(c("lat", "long") %in% colnames(df)) == 2) {  
 alert(title = "Separator Error !!",  
 text = "It seems that you have chosen a wrong seperator. You see can the dataset below the map and choose a corresponding separator.",  
 type = "error")  
   
 }  
 else if (!sum(check.numeric(df$long) + check.numeric(df$lat)) == len \* 2) {  
 alert(title = "Longitude or Latitude Class Error !!",  
 text = "It seems that some of your values in the long or lat columns are not numeric. Double check their formats.",  
 type = "warning")  
 }  
 else if (!sum(df$long <= 180 & df$long >= -180) == len) {  
 alert(title = "Longitude Error !!",  
 text = "It seems that your long column contains a value outside of the allowed range for longitude (-180 - 180). Ensure that all values are within the range.",  
 type = "error")  
 }  
   
 else if (!sum(df$lat <= 90 & df$lat >= -90) == len) {  
 alert(title = "Latitude Error !!",  
 text = "It seems that your lat column contains a value outside of the allowed range for latitude (-90 - 90). More often that not, this is caused by flipping long and lat columns. Ensure that all values are within the range.",  
 type = "error")  
 }  
 else if (ncol(select\_if(df, names(df) %in% label\_col)) != 1) {  
 alert(title = "Label Error !!",  
 text = "It seems that your column name for labelling/describing locations does not correspond to the default: Description. You can change it in the Upload options in the Label name box.",  
 type = "error")  
 }  
 else {  
 alert(title = "Success !!",  
 text = "Your data is loaded in the right format, now you can customize the aesthetics!",  
 type = "success")  
 }  
}

Ideally, we want to show a map at all times in the app. The map should reflect what valid data it has available. We start by defining the base map, that does not have any positional markers. We shows this map if no data has been uploaded yet. The map has selectable tile layers provided by Esri (CITATION NEEDED), a customizable title, a minimap, and a measuring tool. Below is the the code used for defining the base map. when we call input$SOMETHING, we refer to customizable variables available to the user such as the title and its color.

#initiate map  
start\_map <- leaflet()  
#grab all esri tile layers  
esri <- grep("^Esri", providers, value = TRUE)  
#add esri tile layers  
for (provider in esri) {  
 start\_map <-  
 start\_map %>%  
 addProviderTiles(provider,  
 group = provider,  
 options = providerTileOptions(noWrap = !input$wrap\_map))  
}  
  
#use html code to format the title  
map\_title <-  
 paste(  
 "<b style='color:",  
 input$title\_color,  
 ";font-size:",  
 input$title\_size  
 ,  
 "px;font-family:Comic Sans MS'>",  
 #Comic Sans FTW  
 input$map\_title  
 ,  
 "<b/>",  
 sep = ""  
 )  
  
#create the basic map without data  
plot <- start\_map %>%  
 #add layer selector  
 addLayersControl(baseGroups = names(esri),  
 options = layersControlOptions(collapsed = T)) %>%  
 #add minimap  
 addMiniMap(tiles = esri[[1]],  
 toggleDisplay = TRUE,  
 position = "bottomright") %>%  
 #add measuring tool  
 addMeasure(  
 position = "bottomright",  
 primaryLengthUnit = "meters",  
 primaryAreaUnit = "sqmeters",  
 activeColor = "#3D535D",  
 completedColor = "#7D4479"  
 ) %>%  
 #add title  
 addControl(map\_title, "bottomleft")

To avoid crashing the app by attempting to define a map with bad coordinates or something similar, we have implemented a logic switching system that only maps coordinates when they have been approved and only attempts to add images when they have been uploaded. The logic follows the flowchart below:



Flowchart of the mapping logic

Below is the code for the function we use to approve the data format:

#reactive value for approving the data format  
 format\_approved <- reactive ({  
 if (is.null(input$file1)) {  
 return ()  
 }  
 df = filedata()  
 len = length(df[, 1])  
 #format requirements  
 if (sum(c("lat", "long", input$description) %in% colnames(df)) == 3 &  
 sum(check.numeric(df$long) + check.numeric(df$lat)) == len \* 2 &  
 sum(df$long <= 180 &  
 df$long >= -180) == len &  
 sum(df$lat <= 90 & df$lat >= -90) == len) {  
 TRUE  
 }  
 else {  
 FALSE  
 }  
 })

If the data has been approved, we can start defining the map with the markers. We use functions that stem from then *awesome-markers* Leaflet plugin (Voogdt 2014). They make have extended functionality compared to the normal markers, and they make it easy to implement custom icons. The markers and their icons are defined according to e custom settings of the user controlled in the UI. To allow users to format the labels of the markers, users can choose to parse their labels as HTML. Ideally, we would want to find be able to cluster the points close to each other, but clustering breaks our system for showing popup images. To streamline the project and avoid user confusion, we chose not to cluster point, even when no images have been uploaded. Below is the code for defining the map with the markers:

#Create the map with markers if the data has been uploaded and approved  
# if the format has been approved:  
if (req(format\_approved()) == TRUE) {  
 #name of the label column  
   
 icons <- makeAwesomeIcon(  
 text = fa(input$icon),  
 markerColor = input$markerColor,  
 iconColor = input$icon\_color,  
 spin = input$icon\_spin,  
 squareMarker = input$square\_markers  
 )  
 #render labels with html  
 if (input$html == T) {  
 html\_labels <- lapply(df[input$description][, 1], HTML)  
 }  
 else{  
 html\_labels <- df[input$description][, 1]  
 }  
   
 #add markers to plot  
 marker\_plot <- plot %>%  
 addAwesomeMarkers(  
 data = df,  
 lng = df$long,  
 lat = df$lat,  
 label = html\_labels,  
 icon = icons,  
 group = "pnts",  
 #clusterOptions = markerClusterOptions(),  
 options = markerOptions(opacity = 0.8)  
 )  
}

Because we created the map with the markers before, the code for defining the map with images and choosing the right map. If the conditions are met, we add images to plot with markers and show that, else if only the data has been approved, we show the map only with markers, else we show the bare map without markers. Below is the code for deploying the right map:

#add images if imges has been uploaded  
 if (req(format\_approved()) == TRUE & !is.null(df) & !is.null(df$datapath)) {  
 return(marker\_plot %>%  
 addPopupImages(df$datapath, "pnts", 150))  
 }  
 #else return map just with markers  
 else if (req(format\_approved()) == TRUE) {  
 return(marker\_plot)  
 } #otherwise just return the base map  
 else {  
 return(plot)  
 }

If the user choose to export the map, we use the *mapshot()* function from the *mapview* package (Appelhans et al. 2020) to capture the state of the map and save it in the chosen format. The user can either save it as a PNG still image or selfcontained interactive HTML file. There are limits to how much can be stored in the self-contained HTML file, so we restict the image size to be 150 pixels wide. There are still limits to how many images the HTML can handle, but we have not found it. If the problem should arise, the user can still export the map without the images, and just use the markers. Below is our code for exporting the map:

#reactive object for saving the state of the map  
mymap <- reactive({  
 # call the foundational Leaflet map  
 main\_map() %>%  
 # store the view based on UI  
 setView(  
 lng = input$map\_center$lng,  
 lat = input$map\_center$lat,  
 zoom = input$map\_zoom  
 ) %>%  
 showGroup(group = input$map\_groups)  
   
})  
#download map as png saved as the state of the map  
output$downloadPlotPNG <- downloadHandler(  
 filename = "data.png",  
 content = function(file) {  
 mapshot(  
 x = mymap(),  
 file = file,  
 cliprect = "viewport",  
 selfcontained = T  
 )  
 }  
)  
#save html version of the map. The map should be self contained in the html to work wethink...  
output$downloadPlotHTML <- downloadHandler(  
 filename = paste0(getwd(), "/map.html"),  
 content = function(file) {  
 mapshot(  
 x = mymap(),  
 cliprect = "viewport",  
 selfcontained = T,  
 url = file  
 )  
 }  
)

## References

Appelhans, Tim, Florian Detsch, Christoph Reudenbach, and Stefan Woellauer. 2020. *Mapview: Interactive Viewing of Spatial Data in R*. Manual. <https://CRAN.R-project.org/package=mapview>.

Kahle, David, and Hadley Wickham. 2013. “Ggmap: Spatial Visualization with Ggplot2.” *The R Journal* 5 (1): 144–61. <https://journal.r-project.org/archive/2013-1/kahle-wickham.pdf>.

Tennekes, Martijn. 2018. “Tmap: Thematic Maps in R.” *Journal of Statistical Software* 84 (6): 1–39. <https://doi.org/10.18637/jss.v084.i06>.

Voogdt, lennard. 2014. *Leaflet.Awesome-Markers*. <https://github.com/lennardv2/Leaflet.awesome-markers>.

Wickham, Hadley. 2016. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. <https://ggplot2.tidyverse.org>.