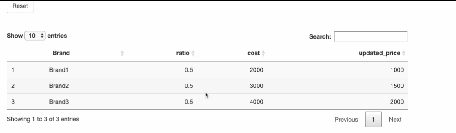
Tables are very much the standard way of representing data in dashboard along with visualizations. Wouldnt it be more useful if you could edit the values in the tables to trigger some calculations and update the values on the fly . These can be used for adjusting allocations or budgets in a project.



**Libraries**

The libraries which we will be using are shiny for the app itself, dplyr and DT for displaying and editing the tables.

library(shiny)

library(dplyr)

library(DT)

**Data**

For demo purpose we are creating a dataframe with three brands and few values.

input\_data <- data.frame(Brand = c("Brand1", "Brand2","Brand3"),

ratio = c (.5, .5, .5),

cost = c(2000, 3000, 4000),

stringsAsFactors = FALSE) %>%

mutate(updated\_price = cost \* ratio)

**Editing function/module**

One thing I have understood over time while building shiny apps is the importance of modules. As the functionalities in the app keep on increasing it becomes difficult to maintain the app in server.r. So I have tried to write the editing function in a module modFunction . You can call this module from the server function of the app.

modFunction <- function(input, output, session, data,reset) {

v <- reactiveValues(data = data)

proxy = dataTableProxy("mod\_table")

observeEvent(input$mod\_table\_cell\_edit, {

print(names(v$data))

info = input$mod\_table\_cell\_edit

str(info)

i = info$row

j = info$col

k = info$value

str(info)

isolate(

if (j %in% match(c("ratio","cost","updated\_price"), names(v$data))) {

print(match(c("ratio","cost", "updated\_price"), names(v$data)))

v$data[i, j] <<- DT::coerceValue(k, v$data[i, j])

print(v$data)

if (j %in% match("cost", names(v$data))) {

v$data$updated\_price <<- v$data$cost \* v$data$ratio

}

if (j %in% match("ratio", names(v$data))) {

v$data$updated\_price <<- v$data$cost \* v$data$ratio

}

} else {

stop("You cannot change this column.") # check to stop the user from editing only few columns

}

)

replaceData(proxy, v$data, resetPaging = FALSE) # replaces data displayed by the updated table

})

### Reset Table

observeEvent(reset(), {

v$data <- data # your default data

})

print(isolate(colnames(v$data)))

output$mod\_table <- DT::renderDataTable({

DT::datatable(v$data, editable = TRUE)

})

}

So if you see the code above once the user edits the ratio or cost it updates the updated\_price . You can also allow the user to modify only a few columns as you see above. If the user tries to edit Brand it would throw an error.If the user feels the values in the tables once they are modified doesnt seem to be correct they can click on the reset to get the values in the table to default.

Below is the actual code for the shiny app which calls the edit table function modFunction. One other advantage of using the modules it decreases the code to be maintained in the shiny app itself.

shinyApp(

ui = basicPage(

mainPanel(

actionButton("reset", "Reset"),

tags$hr(),

modFunctionUI("editable")

)

),

server = function(input, output) {

demodata<-input\_data

callModule(modFunction,"editable", demodata,

reset = reactive(input$reset))

}

)

**Code**

You can find code for the app [here](https://gist.github.com/senthilthyagarajan/a714446c9acd8127e4cbef6ede09b1fc).