There are times when it costs more than it should to leverage javascript, database, html, models and algorithms in one language. Now maybe is time for connecting some dots, without stretching too much.

* If you have been developing shiny apps, consider letting it sit on one live database instead of manipulating data I/O by hand?
* If you use DT to display tables in shiny apps, care to unleash the power of interactivity to its full?
* If you struggle with constructing SQL queries in R, so did we.

Inspired (mainly) by the exciting new inline editing feature of [DT](https://blog.rstudio.com/2018/03/29/dt-0-4/), we created a minimal shiny app demo to show how you can update multiple values from DT and send the edits to database at a time.

As seen in the screenshot, after double clicking on a cell and editing the value, Save and Cancel buttons will show up. Continue editing, the updates are stored in a temporary (reactiveValue) object. Click on Save if you want to send bulk updates to database; click on Cancel to reset.

**Global**

On the global level, we use pool to manage database connections. A database connection pool object is constructed. With the onStop() function, the pool object gets closed after a session ends. It massively saves you from worrying about when to open or close a connection.

# Define pool handler by pool on global level

pool <- pool::dbPool(drv = dbDriver("PostgreSQL"),

dbname="demo",

host="localhost",

user= "postgres",

password="ava2post")

onStop(function() {

poolClose(pool)

}) # important!

The next job is to define a function to update database. The glue\_sql function puts together a SQL query in a human readable way. Writing SQL queries in R was bit of a nightmare. If you used to assemble a SQL clause by sprintf or past, you know what I’m talking about. The glued query is then processed by sqlInterpolate for SQL injection protection before being executed.

updateDB <- function(editedValue, pool, tbl){

# Keep only the last modification for a cell

editedValue <- editedValue %>%

group\_by(row, col) %>%

filter(value == dplyr::last(value)| [is.na](http://is.na)(value)) %>%

ungroup()

conn <- poolCheckout(pool)

lapply(seq\_len(nrow(editedValue)), function(i){

id = editedValue$row[i]

col = dbListFields(pool, tbl)[editedValue$col[i]]

value = editedValue$value[i]

query <- glue::glue\_sql("UPDATE {`tbl`} SET

{`col`} = {value}

WHERE id = {id}

", .con = conn)

dbExecute(conn, sqlInterpolate(ANSI(), query))

})

poolReturn(conn)

print(editedValue)

return(invisible())

}

**Server**

We begin with server.R from defining a couple of reactive values: **data** for most dynamic data object, **dbdata** for what’s in database, **dataSame** for whether data has changed from database, **editedInfo** for edited cell information (row, col and value). Next, create a reactive expression of source data to retrieve data, and assign it to reactive values.

# Generate reactive values

rvs <- reactiveValues(

data = NA,

dbdata = NA,

dataSame = TRUE,

editedInfo = NA

)

# Generate source via reactive expression

mysource <- reactive({

pool %>% tbl("nasa") %>% collect()

})

# Observe the source, update reactive values accordingly

observeEvent(mysource(), {

# Lightly format data by arranging id

# Not sure why disordered after sending UPDATE query in db

data <- mysource() %>% arrange(id)

rvs$data <- data

rvs$dbdata <- data

})

We then render a DataTable object, create its proxy. Note that the **editable** parameter needs to be explicitly turned on. Finally with some format tweaking, we can merge the cell information, including row id, column id and value, with DT proxy and keep all edits as a single reactive value.

# Render DT table and edit cell

#

# no curly bracket inside renderDataTable

# selection better be none

# editable must be TRUE

output$mydt <- DT::renderDataTable(

rvs$data, rownames = FALSE, editable = TRUE, selection = 'none'

)

proxy3 = dataTableProxy('mydt')

observeEvent(input$mydt\_cell\_edit, {

info = input$mydt\_cell\_edit

i = info$row

j = info$col = info$col + 1 # column index offset by 1

v = info$value

info$value <- as.numeric(info$value)

rvs$data[i, j] <<- DT::coerceValue(v, purrr::flatten\_dbl(rvs$data[i, j]))

replaceData(proxy3, rvs$data, resetPaging = FALSE, rownames = FALSE)

rvs$dataSame <- identical(rvs$data, rvs$dbdata)

if (all([is.na](http://is.na)(rvs$editedInfo))) {

rvs$editedInfo <- data.frame(info)

} else {

rvs$editedInfo <- dplyr::bind\_rows(rvs$editedInfo, data.frame(info))

}

})

Once Save button is clicked upon, send bulk updates to database using the function we defined above. Discard current edits and revert DT to last saved status of database when you hit Cancel. Last chunk is a little trick that generates interactive UI buttons. When dynamic data object differs from the database representative object, show Save and Cancel buttons; otherwise hide them.

# Update edited values in db once save is clicked

observeEvent(input$save, {

updateDB(editedValue = rvs$editedInfo, pool = pool, tbl = "nasa")

rvs$dbdata <- rvs$data

rvs$dataSame <- TRUE

})

# Observe cancel -> revert to last saved version

observeEvent(input$cancel, {

rvs$data <- rvs$dbdata

rvs$dataSame <- TRUE

})

# UI buttons

output$buttons <- renderUI({

div(

if (! rvs$dataSame) {

span(

actionButton(inputId = "save", label = "Save",

class = "btn-primary"),

actionButton(inputId = "cancel", label = "Cancel")

)

} else {

span()

}

)

})

**UI**

The UI part is exactly what you normally do. Nothing new.