**Transcript with GEMINI to help build ‘heatmap’ calendar plot – 4/02/25**

If I feed you some R code that builds a shiny app, can you show me how to build a calendar using heat map like functions in R? First, let me also give you the sql code that builds the database the app queries from.

Sounds like a fun project! I'm ready when you are. Please go ahead and share the R code for your Shiny app and the SQL code for your database. Once I have those, I'll be happy to help you figure out how to integrate a calendar heatmap using R. I'm looking forward to seeing what you've built!

SQLlite code: CREATE TABLE own (

  o\_id VARCHAR(255) NOT NULL,

  name VARCHAR(255) NOT NULL,

  phone VARCHAR(255) NOT NULL,

  address VARCHAR(255) NOT NULL,

  PRIMARY KEY (o\_id)

);

CREATE TABLE staff (

  s\_id VARCHAR(255) NOT NULL,

  name VARCHAR(255) NOT NULL,

  salary INTEGER NOT NULL,

  position VARCHAR(255) NOT NULL,

  PRIMARY KEY (s\_id)

);

CREATE TABLE animal (

  aid VARCHAR(255) NOT NULL,

  name VARCHAR(255) NOT NULL,

  age INTEGER NOT NULL,

  breed VARCHAR(255),

  sex VARCHAR(255) NOT NULL,

  species VARCHAR(255) NOT NULL,

  o\_id VARCHAR(255) NOT NULL,

  PRIMARY KEY (aid),

  FOREIGN KEY (o\_id) REFERENCES own(o\_id)

);

CREATE TABLE vital (

  r\_id VARCHAR(255) NOT NULL,

  weight NUMERIC(5,2),

  rbc NUMERIC(5,2),

  wbc NUMERIC(5,2),

  respiration NUMERIC(5,2),

  ph NUMERIC(5,2),

  pulse NUMERIC(5,2),

  glucose NUMERIC(5,2),

  s\_id VARCHAR(255) NOT NULL,

  aid VARCHAR(255) NOT NULL,

  PRIMARY KEY (r\_id),

  FOREIGN KEY (s\_id) REFERENCES staff(s\_id),

  FOREIGN KEY (aid) REFERENCES animal(aid)

);

CREATE TABLE appointment (

  appointment\_date DATE NOT NULL,

  appointment\_time VARCHAR NOT NULL,

  reason VARCHAR(255),

  ontime VARCHAR(255),

  s\_id VARCHAR(255) NOT NULL,

  aid VARCHAR(255) NOT NULL,

  o\_id VARCHAR(255) NOT NULL,

  r\_id VARCHAR(255) NOT NULL,

  PRIMARY KEY (appointment\_date, appointment\_time),

  FOREIGN KEY (s\_id) REFERENCES staff(s\_id),

  FOREIGN KEY (aid) REFERENCES animal(aid),

  FOREIGN KEY (o\_id) REFERENCES own(o\_id),

  FOREIGN KEY (r\_id) REFERENCES vital(r\_id)

);

CREATE TABLE animal\_vet (

  aid VARCHAR(255) NOT NULL,

  s\_id VARCHAR(255) NOT NULL,

  PRIMARY KEY (aid, s\_id),

  FOREIGN KEY (aid) REFERENCES animal(aid),

  FOREIGN KEY (s\_id) REFERENCES staff(s\_id)

);

CREATE TABLE owner\_vet (

  o\_id VARCHAR(255) NOT NULL,

  s\_id VARCHAR(255) NOT NULL,

  PRIMARY KEY (o\_id, s\_id),

  FOREIGN KEY (o\_id) REFERENCES own(o\_id),

  FOREIGN KEY (s\_id) REFERENCES staff(s\_id)

);

Thanks! This gives me a good understanding of your database structure. It looks like the appointment table with the appointment\_date column is the key to building our calendar heatmap.

Now, whenever you're ready, please share the R code for your Shiny app. Once I see how your app is structured and how you're connecting to this database, I can give you more specific guidance on integrating the calendar heatmap. I'm excited to help you bring this visualization to life!

Rcode, I’ve already set up the UI section too:

library(shiny)

library(shinydashboard) # For layout/effects

library(RSQLite)

library(DBI)

library(ggplot2)

library(plotly) # For 'biometric plot' and 'calendar'

library(RPostgres)

library(dplyr) # For 'calendar'

library(lubridate) # For 'calendar'

library(igraph) # For 'network graph'

library(ggraph) # For 'network graph'

# UI

ui <- dashboardPage(

  dashboardHeader(title = "Veterinary Clinic Dashboard"),

  skin = "purple",

  # Setting up collapsible sidebar:

  dashboardSidebar(

    sidebarMenu(

      menuItem("Tabular Views", tabName = "tables", icon = icon("clipboard")),

      menuItem("Compare Biometrics", tabName = "plots", icon = icon("chart-line")),

      menuItem("View Appointments", tabName = "appoints", icon = icon("calendar")),

      menuItem("Pet-Owner Network", tabName = "network", icon = icon("sitemap"))

    ),

    # Adding bottom-text to sidebar

    tags$div(

      style = "margin-top: 20px; margin-left: 10px; font-size: 12px; color: #888;",

      HTML("EPPS 6354.001 - Final Project App. Numbers are simulated from ranges bound

           by findings in the literature.<br>",

           "<a href='https://tylermerrill.github.io' target='\_blank'>For details, click to read project report.</a>")

    )

  ),

  dashboardBody(

    # Setting up the database tables:

    tabItems(

      tabItem(tabName = "tables",

              fluidRow(

                tabsetPanel(

                  id = 'dataset',

                  # One tab for each relation

                  tabPanel("Animals", DT::dataTableOutput("mytable1")),

                  tabPanel("Owners", DT::dataTableOutput("mytable2")),

                  tabPanel("Veterinarians", DT::dataTableOutput("mytable3")),

                  tabPanel("Appointments", DT::dataTableOutput("mytable4")),

                  tabPanel("Vital Records", DT::dataTableOutput("mytable5"))

                )

              )),

      # Set up the biometric plot:

      tabItem(tabName = "plots",

              box(title = "Compare Biometrics", width = 12, # Adjust width as needed

                  fluidRow(

                    # Let user choose selected bio-metrics (can add more options later)

                    column(width = 6, selectInput("x\_biometric", "Select X-axis:", choices = c("weight", "pulse", "ph", "glucose"))),

                    column(width = 6, selectInput("y\_biometric", "Select Y-axis:", choices = c("pulse", "weight", "ph", "glucose"), selected = "pulse")),

                    column(width = 6, selectInput("x1\_biometric", "Select Color Element:", choices = c("weight", "pulse", "ph", "glucose"))),

                    column(width = 6, selectInput("y1\_biometric", "Select Size Element:", choices = c("pulse", "weight", "ph", "glucose")))

                  ),

                  plotlyOutput("scatterPlot")

              )),

      # Set up the calendar:

      tabItem(tabName = "appoints",

              fluidRow(

                box(title = "Appointment Calendar", width = 12,

                    plotlyOutput("calendarPlot"))

              )),

      # Set up the network grpah:

      tabItem(tabName = "network",

              fluidRow(

                box(title = "Owner-Pet Network", width = 12,

                    plotOutput("ownerAnimalNetwork"))

              ))

    )

  )

)

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

  # SQLite connection

  sqlite\_conn <- dbConnect(RSQLite::SQLite(), dbname ='proj4.db') #.db file for my project

  # Table Output

  # I am grabbing all attributes for each relation

  output$mytable1 <- DT::renderDT({

    query <- paste0("SELECT \* ",

                      "FROM animal")

    data <- dbGetQuery(sqlite\_conn, query)

    data

  })

  output$mytable2 <- DT::renderDT({

    query <- paste0("SELECT \* ",

                    "FROM own")

    data <- dbGetQuery(sqlite\_conn, query)

    data

  })

  output$mytable3 <- DT::renderDT({

    query <- paste0("SELECT \* ",

                    "FROM staff")

    data <- dbGetQuery(sqlite\_conn, query)

    data

  })

  output$mytable4 <- DT::renderDT({

    query <- paste0("SELECT \* ",

                    "FROM appointment")

    data <- dbGetQuery(sqlite\_conn, query)

    data

  })

  output$mytable5 <- DT::renderDT({

    query <- paste0("SELECT \* ",

                    "FROM vital")

    data <- dbGetQuery(sqlite\_conn, query)

    data

  })

  # Reactive expression for the scatter plot

  # Get all attributes from 'vital'

  output$scatterPlot <- renderPlotly({

    query <- paste0("SELECT \* ",

                    "FROM vital")

    data <- dbGetQuery(sqlite\_conn, query)

    data

    # Add input fields

    x\_var <- input$x\_biometric

    y\_var <- input$y\_biometric

    x1\_var <- input$x1\_biometric

    y1\_var <- input$y1\_biometric

    # If data is selected (not null), proceed to use values of fields above

    if (!is.null(data) && !is.null(x\_var) && !is.null(y\_var) && x\_var != "" && y\_var != "") {

      # Then, plot it

      p <- ggplot(data, aes\_string(x = x\_var, y = y\_var, color = x1\_var, size = y1\_var)) +

        geom\_point() +

        theme\_minimal() +

        labs(title = paste(y\_var, "vs.", x\_var),

             x = x\_var,

             y = y\_var) +

        theme(text = element\_text(family = "Palatino"))

      ggplotly(p)

    } else {

      # Display a message if nothing is selected (hopefully never)

      plotly() %>% add\_trace(x = 1, y = 1, text = "Select metrics to display plot", mode = "text")

    }

  })

Great! Thanks for sharing your Shiny app code. You've got a well-structured dashboard with several interesting tabs. Now, let's focus on building that calendar heatmap within your "View Appointments" tab.

Here's a step-by-step approach to integrate a calendar heatmap using the plotly and dplyr packages, which you've already included in your dependencies:

**1. Query and Prepare Appointment Data:**

First, within your server function, you'll need to query the appointment table to get the dates of the appointments. Then, you'll prepare this data for the heatmap.

Code snippet

# Data for the calendar heatmap

appointment\_data <- reactive({

query <- "SELECT appointment\_date FROM appointment"

data <- dbGetQuery(sqlite\_conn, query) %>%

mutate(date = as.Date(appointment\_date)) %>%

group\_by(date) %>%

summarise(count = n()) %>%

ungroup() %>%

mutate(

year = year(date),

month = month(date),

day = day(date)

)

return(data)

})

In this reactive expression:

* We query the appointment\_date from your appointment table.
* We convert the appointment\_date to a Date object using as.Date().
* We group the data by date and count the number of appointments on each day using summarise(count = n()).
* We extract the year, month, and day for easier plotting.

**2. Create the Calendar Heatmap with plotly:**

Now, you can use the prepared data to create the calendar heatmap within your renderPlotly for the calendarPlotoutput.

Code snippet

output$calendarPlot <- renderPlotly({

app\_data <- appointment\_data()

if (nrow(app\_data) > 0) {

plot\_ly(app\_data,

x = ~day,

y = ~as.factor(month),

z = ~count,

type = "heatmap",

colors = "viridis", # You can choose other color scales

text = ~paste0(format(date, "%Y-%m-%d"), "<br>", count, " appointments"),

hoverinfo = "text"

) %>%

facet\_grid(rows = ~year, switch = "y") %>%

layout(

title = "Appointment Calendar Heatmap",

xaxis = list(title = "Day"),

yaxis = list(title = "Month", categoryorder = "array", categoryarray = month.abb),

legend = list(title = "Appointment Count")

)

} else {

plotly() %>% add\_trace(x = 1, y = 1, text = "No appointment data available", mode = "text")

}

})

In this renderPlotly:

* We access the processed appointment data from appointment\_data().
* We use plot\_ly() to create the heatmap:
  + x is set to the day of the month.
  + y is set to the month (as a factor to control order).
  + z represents the count of appointments, which will determine the color intensity.
  + type = "heatmap" specifies the plot type.
  + colors = "viridis" sets the color scale. You can explore other plotly color scales.
  + text creates the hover information to show the date and appointment count.
  + hoverinfo = "text" ensures only the custom text is displayed on hover.
* facet\_grid(rows = ~year, switch = "y") separates the heatmap by year.
* layout() customizes the plot title, axis labels, and month order.

**3. Integrate into the UI:**

You've already set up the plotlyOutput("calendarPlot") in your UI within the "View Appointments" tab, so no changes are needed there.

**Complete server Function with Calendar Heatmap:**

Here's how your updated server function would look with the calendar heatmap integration:

Code snippet

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

# SQLite connection

sqlite\_conn <- dbConnect(RSQLite::SQLite(), dbname ='proj4.db')

# Table Output

output$mytable1 <- DT::renderDT({

query <- paste0("SELECT \* FROM animal")

dbGetQuery(sqlite\_conn, query)

})

output$mytable2 <- DT::renderDT({

query <- paste0("SELECT \* FROM own")

dbGetQuery(sqlite\_conn, query)

})

output$mytable3 <- DT::renderDT({

query <- paste0("SELECT \* FROM staff")

dbGetQuery(sqlite\_conn, query)

})

output$mytable4 <- DT::renderDT({

query <- paste0("SELECT \* FROM appointment")

dbGetQuery(sqlite\_conn, query)

})

output$mytable5 <- DT::renderDT({

query <- paste0("SELECT \* FROM vital")

dbGetQuery(sqlite\_conn, query)

})

# Reactive expression for the scatter plot

output$scatterPlot <- renderPlotly({

query <- paste0("SELECT \* FROM vital")

data <- dbGetQuery(sqlite\_conn, query)

x\_var <- input$x\_biometric

y\_var <- input$y\_biometric

x1\_var <- input$x1\_biometric

y1\_var <- input$y1\_biometric

if (!is.null(data) && !is.null(x\_var) && !is.null(y\_var) && x\_var != "" && y\_var != "") {

p <- ggplot(data, aes\_string(x = x\_var, y = y\_var, color = x1\_var, size = y1\_var)) +

geom\_point() +

theme\_minimal() +

labs(title = paste(y\_var, "vs.", x\_var), x = x\_var, y = y\_var) +

theme(text = element\_text(family = "Palatino"))

ggplotly(p)

} else {

plotly() %>% add\_trace(x = 1, y = 1, text = "Select metrics to display plot", mode = "text")

}

})

# Data for the calendar heatmap

appointment\_data <- reactive({

query <- "SELECT appointment\_date FROM appointment"

data <- dbGetQuery(sqlite\_conn, query) %>%

mutate(date = as.Date(appointment\_date)) %>%

group\_by(date) %>%

summarise(count = n()) %>%

ungroup() %>%

mutate(

year = year(date),

month = month(date),

day = day(date)

)

return(data)

})

# Calendar Heatmap Output

output$calendarPlot <- renderPlotly({

app\_data <- appointment\_data()

if (nrow(app\_data) > 0) {

plot\_ly(app\_data,

x = ~day,

y = ~as.factor(month),

z = ~count,

type = "heatmap",

colors = "viridis",

text = ~paste0(format(date, "%Y-%m-%d"), "<br>", count, " appointments"),

hoverinfo = "text"

) %>%

facet\_grid(rows = ~year, switch = "y") %>%

layout(

title = "Appointment Calendar Heatmap",

xaxis = list(title = "Day"),

yaxis = list(title = "Month", categoryorder = "array", categoryarray = month.abb),

legend = list(title = "Appointment Count")

)

} else {

plotly() %>% add\_trace(x = 1, y = 1, text = "No appointment data available", mode = "text")

}

})

# Network Graph Output (assuming you'll implement this later)

# output$ownerAnimalNetwork <- renderPlot({

# # Your network graph code here

# })

# Close the database connection when the session ends

session$onSessionEnded(function() {

dbDisconnect(sqlite\_conn)

})

}

shinyApp(ui, server)

**Before Running:**

* **Ensure your proj4.db file is in the correct location** where your Shiny app can access it.
* **Make sure you have all the necessary packages installed** (shiny, shinydashboard, RSQLite, DBI, ggplot2, plotly, dplyr, lubridate, igraph, ggraph). You can install them using install.packages("package\_name").

Now, when you run your Shiny app and navigate to the "View Appointments" tab, you should see a calendar heatmap visualizing the number of appointments on each day. The intensity of the color will indicate the appointment count. Each year will be displayed as a separate row.

Let me know if you encounter any issues or if you'd like to further customize the appearance or functionality of the calendar heatmap!

Can we try using other libraries to accomplish the same thing?

Yes, absolutely! While plotly is excellent for interactive heatmaps, we can definitely explore other R libraries to create calendar heatmaps. Here are a couple of popular alternatives and how you might implement them:

**1. Using ggplot2 with geom\_tile():**

The ggplot2 library is a powerful and flexible choice for creating static graphics. We can achieve a calendar heatmap effect by using geom\_tile() to draw colored rectangles for each day.

Code snippet

output$calendarPlot <- renderPlotly({ # Changed to renderPlot for ggplot

app\_data <- appointment\_data()

if (nrow(app\_data) > 0) {

# Create a data frame with all days in the range for proper alignment

min\_date <- min(app\_data$date)

max\_date <- max(app\_data$date)

all\_days <- seq(min\_date, max\_date, by = "day")

calendar\_data <- data.frame(date = all\_days) %>%

mutate(

year = year(date),

month = month(date),

day = day(date)

) %>%

left\_join(app\_data, by = "date") %>%

replace\_na(list(count = 0)) %>%

mutate(

week\_day = wday(date, week\_start = 1), # Monday as the first day of the week

week\_of\_year = isoweek(date)

)

# Adjust week of year for proper calendar layout

calendar\_data <- calendar\_data %>%

mutate(adjusted\_week = ifelse(month(date) == 1 & week\_of\_year > 50, 0, week\_of\_year)) %>%

mutate(plot\_week = adjusted\_week + ifelse(month(date) == 1, 0, max(adjusted\_week[month(date) == 1])))

p <- ggplot(calendar\_data, aes(x = week\_day, y = -plot\_week, fill = count)) +

geom\_tile(color = "white", linewidth = 0.5) +

scale\_fill\_viridis\_c(option = "viridis") + # Choose your color scale

facet\_grid(rows = vars(year), cols = vars(month), switch = "y") +

coord\_equal() +

labs(title = "Appointment Calendar Heatmap",

x = "Day of Week",

y = NULL,

fill = "Appointments") +

scale\_x\_continuous(breaks = 1:7, labels = c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")) +

scale\_y\_continuous(breaks = NULL) +

theme\_minimal() +

theme(

strip.text.y = element\_text(angle = 0, hjust = 1),

axis.ticks = element\_blank(),

panel.spacing.y = unit(0.5, "lines")

)

ggplotly(p) # Keep it as plotly for interactivity

} else {

plotly() %>% add\_trace(x = 1, y = 1, text = "No appointment data available", mode = "text")

}

})

In this ggplot2 approach:

* We first create a calendar\_data frame that includes all days within the appointment date range to ensure a complete calendar structure.
* We calculate the week\_day and week\_of\_year to position the tiles correctly. We also do some adjustments to the week\_of\_year to handle the transition between years correctly in the plot layout.
* geom\_tile() creates the colored rectangles, with the fill aesthetic determined by the count of appointments.
* scale\_fill\_viridis\_c() sets the color gradient.
* facet\_grid(rows = vars(year), cols = vars(month), switch = "y") arranges the plot into a calendar-like grid with years as rows and months as columns.
* coord\_equal() ensures that the tiles are square.
* We customize the labels and theme for better readability.
* Finally, we wrap the ggplot object with ggplotly() to retain interactivity. If you prefer a static image, you can use renderPlot() instead of renderPlotly().

**2. Using the calheatmapR package:**

The calheatmapR package is specifically designed for creating calendar heatmaps and offers a more specialized set of functions and customization options.

First, you'll need to install it: install.packages("calheatmapR").

Then, you can integrate it into your Shiny app:

Code snippet

library(calheatmapR)

# ... (rest of your UI and server setup) ...

output$calendarPlot <- renderCalheatmapR({

app\_data <- appointment\_data()

if (nrow(app\_data) > 0) {

calheatmap(app\_data,

x = "date",

y = "count",

domain = "month",

subdomain = "day",

data.label = TRUE,

label = "Appointments on {x} is {y}",

legend = TRUE,

year.as.row = TRUE # Display years in rows

)

} else {

# You might need a placeholder if there's no data for calheatmapR

# For simplicity, we can return NULL or a message.

return(NULL)

}

})

# In your UI, you'll need to use calheatmapROutput instead of plotlyOutput:

# calheatmapROutput("calendarPlot")

In this calheatmapR approach:

* We directly use the calheatmap() function.
* x = "date" specifies the date variable.
* y = "count" specifies the value to be represented by the heatmap color.
* domain = "month" sets the primary grouping level.
* subdomain = "day" sets the secondary level.
* data.label = TRUE displays the appointment count on each day.
* label customizes the hover text.
* legend = TRUE shows the color legend.
* year.as.row = TRUE arranges the calendar by year in rows.

**Important Considerations:**

* **Interactivity:** plotly inherently provides interactive plots (tooltips, zooming, panning). If you use ggplot2directly with renderPlot(), the plot will be static. Wrapping it with ggplotly() adds interactivity. calheatmapRoften provides some level of interactivity as well.
* **Customization:** Each library offers different levels and ways to customize the appearance of the heatmap (colors, labels, layout, etc.).
* **Dependencies:** Remember to include the necessary library in your library() calls at the beginning of your app.R file.
* **UI Output:** If you switch to calheatmapR, make sure to change the output function in your UI from plotlyOutput("calendarPlot") to calheatmapROutput("calendarPlot").

Feel free to try out these alternative libraries. Each has its strengths and might suit your preferences or specific customization needs better. Let me know if you'd like help implementing either of these in more detail!

Sources and related content

[Chapter 23 Shiny | Data Visualization](https://stulp.gmw.rug.nl/dataviz/shiny.html" \t "_blank)

[stulp.gmw.rug.nl](https://stulp.gmw.rug.nl/dataviz/shiny.html" \t "_blank)

[calheatmapR](https://durtal.github.io/calheatmapR/" \t "_blank)

[durtal.github.io](https://durtal.github.io/calheatmapR/" \t "_blank)

Show me how with the ggplot option

Alright, let's get that ggplot2 calendar heatmap integrated into your Shiny app. Here's the updated code for your server function, specifically focusing on the calendarPlot output using ggplot2 and then making it interactive with plotly:

Code snippet

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

# SQLite connection

sqlite\_conn <- dbConnect(RSQLite::SQLite(), dbname ='proj4.db')

# Table Output (no changes needed here)

output$mytable1 <- DT::renderDT({ /\* ... \*/ })

output$mytable2 <- DT::renderDT({ /\* ... \*/ })

output$mytable3 <- DT::renderDT({ /\* ... \*/ })

output$mytable4 <- DT::renderDT({ /\* ... \*/ })

output$mytable5 <- DT::renderDT({ /\* ... \*/ })

# Reactive expression for the scatter plot (no changes needed here)

output$scatterPlot <- renderPlotly({ /\* ... \*/ })

# Data for the calendar heatmap

appointment\_data <- reactive({

query <- "SELECT appointment\_date FROM appointment"

data <- dbGetQuery(sqlite\_conn, query) %>%

mutate(date = as.Date(appointment\_date)) %>%

group\_by(date) %>%

summarise(count = n()) %>%

ungroup() %>%

mutate(

year = year(date),

month = month(date),

day = day(date)

)

return(data)

})

# Calendar Heatmap Output using ggplot2

output$calendarPlot <- renderPlotly({

app\_data <- appointment\_data()

if (nrow(app\_data) > 0) {

# Create a data frame with all days in the range for proper alignment

min\_date <- min(app\_data$date)

max\_date <- max(app\_data$date)

all\_days <- seq(min\_date, max\_date, by = "day")

calendar\_data <- data.frame(date = all\_days) %>%

mutate(

year = year(date),

month = month(date),

day = day(date)

) %>%

left\_join(app\_data, by = "date") %>%

replace\_na(list(count = 0)) %>%

mutate(

week\_day = wday(date, week\_start = 1), # Monday as the first day of the week

week\_of\_year = isoweek(date)

)

# Adjust week of year for proper calendar layout

calendar\_data <- calendar\_data %>%

mutate(adjusted\_week = ifelse(month(date) == 1 & week\_of\_year > 50, 0, week\_of\_year)) %>%

mutate(

# Calculate a plotting week that restarts at 1 for each year

plot\_week = adjusted\_week + (year(date) - min(year(date))) \* 53 # Assuming max 53 weeks

)

# Create the ggplot object

p <- ggplot(calendar\_data, aes(x = week\_day, y = -plot\_week, fill = count,

text = paste0(format(date, "%Y-%m-%d"), "<br>", count, " appointments"))) +

geom\_tile(color = "white", linewidth = 0.5) +

scale\_fill\_viridis\_c(option = "viridis") + # Choose your color scale

facet\_grid(rows = vars(year), cols = vars(month), switch = "y") +

coord\_equal() +

labs(title = "Appointment Calendar Heatmap",

x = "Day of Week",

y = NULL,

fill = "Appointments") +

scale\_x\_continuous(breaks = 1:7, labels = c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")) +

scale\_y\_continuous(breaks = NULL) +

theme\_minimal() +

theme(

strip.text.y = element\_text(angle = 0, hjust = 1),

axis.ticks = element\_blank(),

panel.spacing.y = unit(0.5, "lines")

)

# Convert ggplot to plotly for interactivity

ggplotly(p, tooltip = "text")

} else {

plotly() %>% add\_trace(x = 1, y = 1, text = "No appointment data available", mode = "text")

}

})

# Network Graph Output (no changes needed here)

# output$ownerAnimalNetwork <- renderPlot({ /\* ... \*/ })

# Close the database connection (no changes needed here)

session$onSessionEnded(function() { /\* ... \*/ })

}

**Key Changes and Explanations:**

1. **plot\_week Calculation:** The calculation for plot\_week is adjusted to try and keep the weeks aligned vertically across different years within each month facet. It now incorporates the year to offset the week numbers. This isn't a perfect calendar layout like calheatmapR, but it arranges the days by week within each month and year.
2. **aes() for ggplot:** Inside ggplot(), we now include the text aesthetic:

Code snippet

aes(x = week\_day, y = -plot\_week, fill = count,

text = paste0(format(date, "%Y-%m-%d"), "<br>", count, " appointments"))

This creates the text that will appear when you hover over each tile in the plotly output.

1. **geom\_tile():** This remains the same, drawing the colored rectangles.
2. **facet\_grid():** This keeps the layout organized by year (rows) and month (columns).
3. **ggplotly():** We wrap the ggplot object p with ggplotly(p, tooltip = "text"). The tooltip = "text"argument tells plotly to use the text aesthetic we defined in ggplot() for the hover information.

**To Run This:**

1. Make sure you have the ggplot2 and plotly packages installed (install.packages(c("ggplot2", "plotly"))).
2. Replace the renderPlotly code for output$calendarPlot in your server function with the code above.
3. Ensure your UI still has plotlyOutput("calendarPlot") in the "View Appointments" tab.
4. Run your Shiny app.

Now, when you navigate to the "View Appointments" tab, you should see a calendar-like grid where each tile represents a day. The color intensity indicates the number of appointments, and when you hover over a tile, you should see the date and the appointment count.

Keep in mind that achieving a perfectly traditional calendar layout with ggplot2 alone can be a bit more involved due to how ggplot2 handles discrete scales and faceting. Libraries specifically designed for calendar heatmaps (like calheatmapR) often handle the layout more elegantly. However, this ggplot2 approach with plotly provides a functional and interactive heatmap representation of your appointment data.

Can we make sure dates are in date format, and make sure the days tiles are squares? Also, the year labels are upside down on the right side of the graph

You're absolutely right! Let's fix those issues: ensuring the date format, making the day tiles square, and correcting the year label orientation.

Here's the updated server function code for the calendarPlot output:

Code snippet

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

# SQLite connection

sqlite\_conn <- dbConnect(RSQLite::SQLite(), dbname ='proj4.db')

# Table Output (no changes needed)

output$mytable1 <- DT::renderDT({ /\* ... \*/ })

output$mytable2 <- DT::renderDT({ /\* ... \*/ })

output$mytable3 <- DT::renderDT({ /\* ... \*/ })

output$mytable4 <- DT::renderDT({ /\* ... \*/ })

output$mytable5 <- DT::renderDT({ /\* ... \*/ })

# Reactive expression for the scatter plot (no changes needed)

output$scatterPlot <- renderPlotly({ /\* ... \*/ })

# Data for the calendar heatmap

appointment\_data <- reactive({

query <- "SELECT appointment\_date FROM appointment"

data <- dbGetQuery(sqlite\_conn, query) %>%

mutate(date = as.Date(appointment\_date)) %>%

group\_by(date) %>%

summarise(count = n()) %>%

ungroup() %>%

mutate(

year = year(date),

month = month(date),

day = day(date)

)

return(data)

})

# Calendar Heatmap Output using ggplot2

output$calendarPlot <- renderPlotly({

app\_data <- appointment\_data()

if (nrow(app\_data) > 0) {

# Create a data frame with all days in the range for proper alignment

min\_date <- min(app\_data$date)

max\_date <- max(app\_data$date)

all\_days <- seq(min\_date, max\_date, by = "day")

calendar\_data <- data.frame(date = all\_days) %>%

mutate(

year = factor(year(date)), # Ensure year is a factor for proper faceting

month = month(date),

day = day(date)

) %>%

left\_join(app\_data, by = "date") %>%

replace\_na(list(count = 0)) %>%

mutate(

week\_day = wday(date, week\_start = 1), # Monday as the first day of the week

week\_of\_year = isoweek(date)

)

# Adjust week of year for proper calendar layout

calendar\_data <- calendar\_data %>%

mutate(adjusted\_week = ifelse(month == 1 & week\_of\_year > 50, 0, week\_of\_year)) %>%

group\_by(year, month) %>%

mutate(plot\_week = adjusted\_week - min(adjusted\_week) + 1) %>%

ungroup()

# Create the ggplot object

p <- ggplot(calendar\_data, aes(x = week\_day, y = -plot\_week, fill = count,

text = paste0(format(date, "%Y-%m-%d"), "<br>", count, " appointments"))) +

geom\_tile(color = "white", linewidth = 0.5) +

scale\_fill\_viridis\_c(option = "viridis") + # Choose your color scale

facet\_grid(rows = vars(year), cols = vars(month), switch = "y") +

coord\_equal() + # Make the tiles square

labs(title = "Appointment Calendar Heatmap",

x = "Day of Week",

y = NULL,

fill = "Appointments") +

scale\_x\_continuous(breaks = 1:7, labels = c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")) +

scale\_y\_continuous(breaks = NULL) +

theme\_minimal() +

theme(

strip.text.y = element\_text(angle = 0, hjust = 1), # Correct year label angle

axis.ticks = element\_blank(),

panel.spacing.y = unit(0.5, "lines")

)

# Convert ggplot to plotly for interactivity

ggplotly(p, tooltip = "text")

} else {

plotly() %>% add\_trace(x = 1, y = 1, text = "No appointment data available", mode = "text")

}

})

# Network Graph Output (no changes needed)

# output$ownerAnimalNetwork <- renderPlot({ /\* ... \*/ })

# Close the database connection (no changes needed)

session$onSessionEnded(function() { /\* ... \*/ })

}

**Here's a breakdown of the changes:**

1. **Date Format:** The appointment\_date is already being converted to a Date object using as.Date() in your appointment\_data reactive expression. The format(date, "%Y-%m-%d") within the aes(text = ...) ensures the date is displayed in a standard year-month-day format on hover. No further explicit date formatting for the plot itself is strictly necessary as ggplot2 and plotly handle Date objects appropriately for plotting.
2. **Square Tiles:** The key to making the tiles square is the coord\_equal() function. We've added this to the ggplot() call:

Code snippet

coord\_equal()

This ensures that the ratio of units on the x and y axes is the same, resulting in square tiles.

1. **Correcting Year Label Orientation:** The year labels are on the right side because of switch = "y" in facet\_grid(). To make them horizontal (not upside down), we need to adjust the theme():

Code snippet

theme(

strip.text.y = element\_text(angle = 0, hjust = 1),

# ... other theme elements

)

* + strip.text.y targets the text labels for the rows (which are the years in this case).
  + angle = 0 sets the rotation angle to 0 degrees (horizontal).
  + hjust = 1 adjusts the horizontal justification so the text is right-aligned against the strip background.

With these changes, your calendar heatmap should now display square day tiles and have the year labels oriented correctly on the right side. Make sure to run your Shiny app with this updated server code!

Can we aggregate data to check for appointments/details after storing our query results, then create a date range for the calendar, then join that with appointment details? I read online the 'lubridate' package might help with this and make it better to use.

Yes, absolutely! That's a more robust approach to ensure you have a complete date range for your calendar and can correctly join it with your appointment data. Using lubridate will indeed make working with dates much easier.

Here's how you can modify the appointment\_data reactive expression in your server function to implement this:

Code snippet

library(lubridate) # Make sure lubridate is loaded

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

# ... (rest of your server code) ...

# Data for the calendar heatmap (with date range and join)

appointment\_data <- reactive({

query <- "SELECT appointment\_date FROM appointment"

appt\_dates\_df <- dbGetQuery(sqlite\_conn, query) %>%

mutate(date = as.Date(appointment\_date))

if (nrow(appt\_dates\_df) > 0) {

# 1. Aggregate appointment counts by date

daily\_counts <- appt\_dates\_df %>%

group\_by(date) %>%

summarise(count = n(), .groups = 'drop')

# 2. Create a full date range

min\_date <- min(daily\_counts$date)

max\_date <- max(daily\_counts$date)

full\_date\_range <- tibble(date = seq(min\_date, max\_date, by = "day")) %>%

mutate(

year = factor(year(date)),

month = month(date),

day = day(date)

)

# 3. Join with appointment details

calendar\_data <- full\_date\_range %>%

left\_join(daily\_counts, by = "date") %>%

replace\_na(list(count = 0)) %>%

mutate(

week\_day = wday(date, week\_start = 1),

week\_of\_year = isoweek(date)

) %>%

group\_by(year, month) %>%

mutate(plot\_week = week\_of\_year - min(week\_of\_year) + 1) %>%

ungroup()

return(calendar\_data)

} else {

# Return an empty tibble with the expected columns if no data

return(tibble(

date = as.Date(character()),

year = factor(character()),

month = integer(),

day = integer(),

count = integer(),

week\_day = integer(),

week\_of\_year = integer(),

plot\_week = integer()

))

}

})

# Calendar Heatmap Output using ggplot2 (no changes needed here)

output$calendarPlot <- renderPlotly({

calendar\_data <- appointment\_data()

if (nrow(calendar\_data) > 0) {

p <- ggplot(calendar\_data, aes(x = week\_day, y = -plot\_week, fill = count,

text = paste0(format(date, "%Y-%m-%d"), "<br>", count, " appointments"))) +

geom\_tile(color = "white", linewidth = 0.5) +

scale\_fill\_viridis\_c(option = "viridis") +

facet\_grid(rows = vars(year), cols = vars(month), switch = "y") +

coord\_equal() +

labs(title = "Appointment Calendar Heatmap",

x = "Day of Week",

y = NULL,

fill = "Appointments") +

scale\_x\_continuous(breaks = 1:7, labels = c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")) +

scale\_y\_continuous(breaks = NULL) +

theme\_minimal() +

theme(

strip.text.y = element\_text(angle = 0, hjust = 1),

axis.ticks = element\_blank(),

panel.spacing.y = unit(0.5, "lines")

)

ggplotly(p, tooltip = "text")

} else {

plotly() %>% add\_trace(x = 1, y = 1, text = "No appointment data available", mode = "text")

}

})

# ... (rest of your server code) ...

}

**Explanation of Changes:**

1. **Initial Query and Date Conversion:** We first fetch the appointment\_date and convert it to a Date object in appt\_dates\_df.
2. **Aggregate Appointment Counts:**

Code snippet

daily\_counts <- appt\_dates\_df %>%

group\_by(date) %>%

summarise(count = n(), .groups = 'drop')

Here, we group the appointment dates by date and use summarise(count = n()) to count the number of appointments on each day. .groups = 'drop' removes the grouping attribute from the result.

1. **Create Full Date Range:**

Code snippet

min\_date <- min(daily\_counts$date)

max\_date <- max(daily\_counts$date)

full\_date\_range <- tibble(date = seq(min\_date, max\_date, by = "day")) %>%

mutate(

year = factor(year(date)),

month = month(date),

day = day(date)

)

We find the minimum and maximum dates from the daily\_counts. Then, seq(min\_date, max\_date, by = "day") creates a sequence of all dates within that range. We then extract the year, month, and day using functions from lubridate (which is implicitly loaded when you load dplyr but it's good practice to explicitly library(lubridate) if you use its functions directly). We also ensure year is a factor for proper faceting.

1. **Join with Appointment Details:**

Code snippet

calendar\_data <- full\_date\_range %>%

left\_join(daily\_counts, by = "date") %>%

replace\_na(list(count = 0)) %>%

mutate(

week\_day = wday(date, week\_start = 1),

week\_of\_year = isoweek(date)

) %>%

group\_by(year, month) %>%

mutate(plot\_week = week\_of\_year - min(week\_of\_year) + 1) %>%

ungroup()

We perform a left\_join() between the full\_date\_range and daily\_counts using the date column. This ensures that every date in the range is present in calendar\_data. replace\_na(list(count = 0)) fills in the count with 0 for days where there were no appointments. The rest of the code calculates the week\_day and plot\_week for the calendar layout, similar to the previous approach.

1. **Handling No Data:** The if (nrow(appt\_dates\_df) > 0) block ensures that we only proceed with creating the date range and joining if there is actual appointment data. If there's no data, it returns an empty tibble with the correct column names to prevent errors in the renderPlotly function.

With this revised appointment\_data reactive expression, you'll have a more complete and accurate calendar heatmap, even for periods with no appointments, as it will include all dates within the range. Remember to have the lubridatepackage loaded in your app.R file.

Thanks, I'll tweak it from here!

Excellent! I'm glad we could get you to this point. Feel free to reach out if you have any more questions or run into any snags as you continue to refine your calendar heatmap. Happy tweaking!