Introduction to R Shiny
Day 1

Posit Implementation Programme

11/09/2023

Public Health Scotland

Learning Outcomes – Day 1

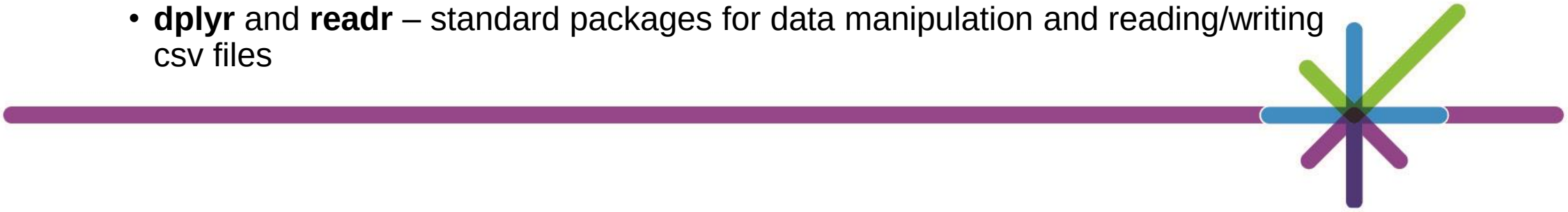
1. Introduction: what is R Shiny?
2. Knowledge of packages relevant to R Shiny work
3. Shiny app breakdown: UI, Server
4. Overall layouts – eg. sidebarLayout (sidebarPanel, mainPanel), navbarPage, tabPanel, fluidRow, column
5. Reading in prepared data
6. Creating charts and tables – ggplot, DT
7. Accessible shinyWidgets and Reactivity: radioButtons and drop-down menus
8. Multi-tab dashboards

What is R Shiny?

- Shiny is an open-source R package where you can create interactive web applications using Posit and R code
- Shiny doesn’t require HTML, CSS or JavaScript knowledge, but can be extended with basic CSS themes, HTML widgets and JavaScript actions
- Shiny works in any R environment and comes with pre-built and customizable output widgets for displaying plots, tables, and printed output of R objects
- **NOTE:** functions associated with Shiny are generally written in CamelCase
- Shiny power - check out an accessible PHS Shiny dashboard: National Therapeutic Indicators

R packages associated with R Shiny work

- There are many packages you may find yourself using when creating a Shiny dashboard!
- Some examples of key packages and their functions:
    - **shiny** – the package that allows all of this to be possible!
    - **shinyWidgets** – for action buttons, drop-downs, radio buttons and many more, check it out: https://shiny.rstudio.com/gallery/widget-gallery.html
    - **shinymanager** – securing and password protecting apps (great for pre-release access dashboards!)
    - **shinyjs** – for bringing basic JavaScript into R eg. enable and disable functions
    - **shinyBS** – create buttons and collapsible panels
    - **shinycssloaders** – for adding icon pictures to tabs
    - **ggplot2**, **plotly**, **DT** – for creating charts/tables
    - **dplyr** and **readr** – standard packages for data manipulation and reading/writing csv files



Shiny app breakdown

- Shiny apps can be contained within a single script (not recommended for full dashboard building) or split across multiple scripts.

- Small apps are contained within a single script called **app.R** which requires three main components to run the app:
  - **user interface object** (defined as UI): layout and appearance of your app.
  - **server function** (defined as server): instructions your computer needs to build the app.
  - A call to the **shinyApp function:** creates Shiny app objects from an explicit UI/server pair.

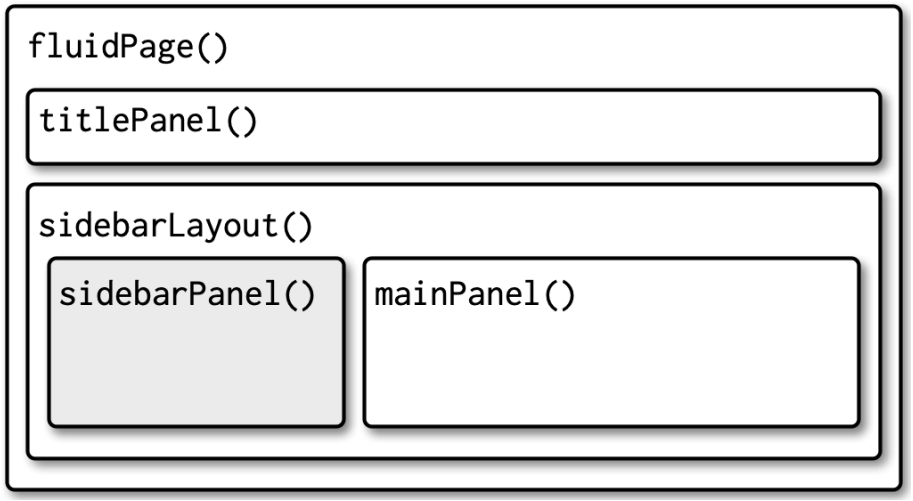
The User Interface (UI)

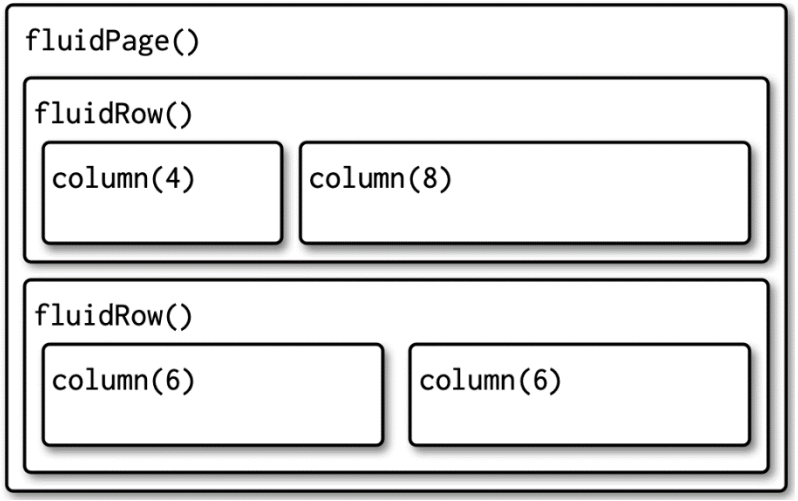
- Controls what is displayed on the application page and how the components are laid out.
- Examples:
    - navigation bars
    - text/titles
    - markdown elements
    - download buttons
    - plot outputs from server
    - user input widgets

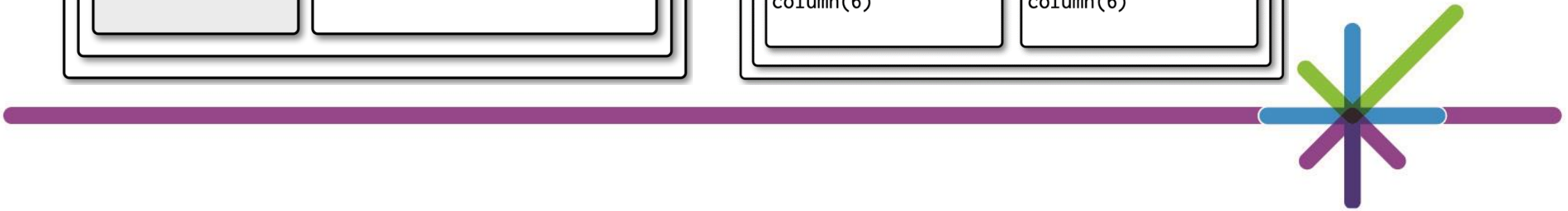
- **Key point:** Shiny uses the function **fluidPage()** to create a display that automatically adjusts to the dimensions of the users browser window. The above UI elements will generally be placed within this function.

The User Interface (UI): Layout

- The UI is built from nested functions specifying the app's layout like a grid of layers
- Filters, buttons, charts and tables are placed within the function for a specific space







The Server

- The server-side controls everything that happens behind the scenes, for example, the data that will be displayed through the UI.

- This part of the script defines how we generate all the plots and tables seen by the user.

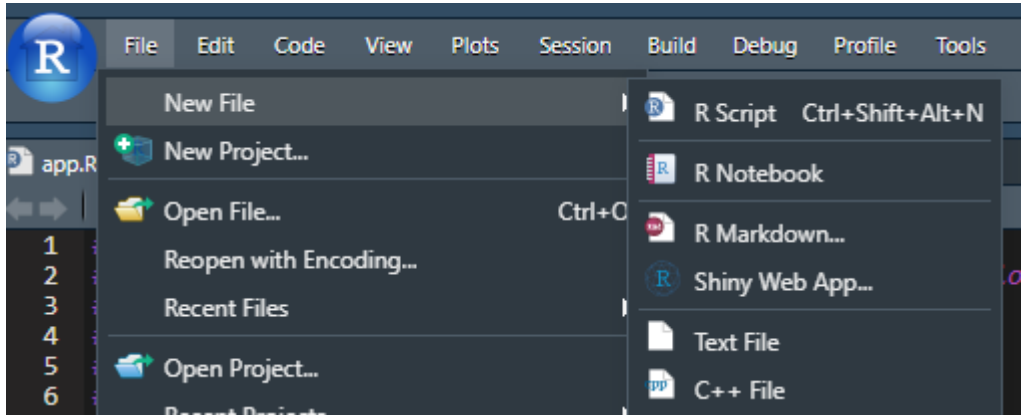
- It also defines how user inputs from our widgets (such as a user selecting from a drop-down menu) affects these plots and tables.

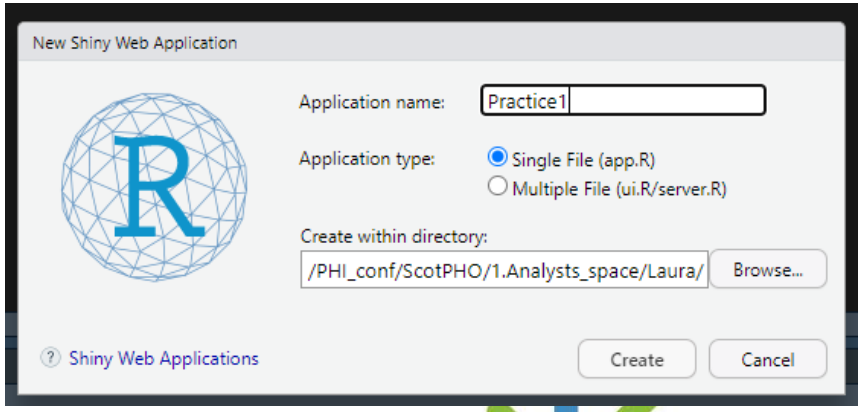
- If necessary, the server section may also be used for minor data wrangling, such as filtering it in such a way that it can be fed into a plot.

Any questions so far?

Code Along: the bare bones of a Shiny app

- Open a new session in Posit.
- Set working directory in stats drive.
- Go to **File** > **New File** > **Shiny Web App**.
- **Name** your app (anything you like eg. “Practice1”) and **set file path** (where you can save your practice scripts).
- Ensure the Application type is set as **Single File (app.R)**.
- Click **Create**.



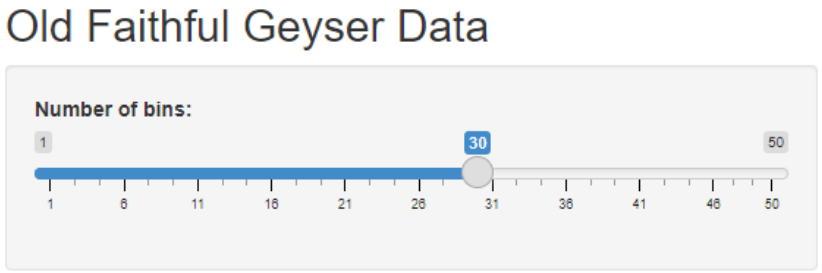


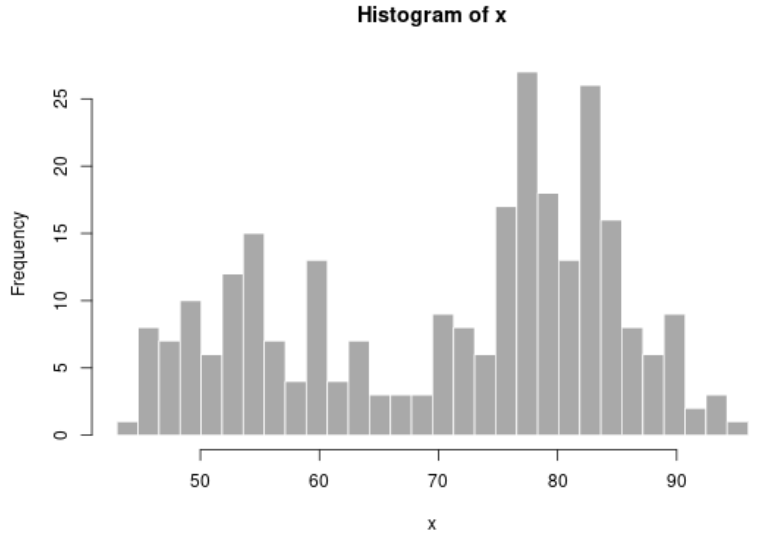
Code Along: the bare bones of a Shiny app

- Upon creating the new Shiny Web App you will see some sample code related to the “Old Faithful Geyser” dataset.

- Click **“Run App”** at the top right of the script window.

- The app may run in a new window, or in the viewer pane (you can change between with the drop-down arrow at “Run App”).





Code Along: Empty Shiny app

- Remove all code relating to Old Faithful Geyser, leaving an empty app.
- The app will run, because it has the three components required; ui, server and a call to the **shinyApp()** function
- This is a good place to start building a new app.

```
library(shiny)

ui <- fluidPage()

server <- function(input, output){
}

shinyApp(ui = ui, server = server)
```

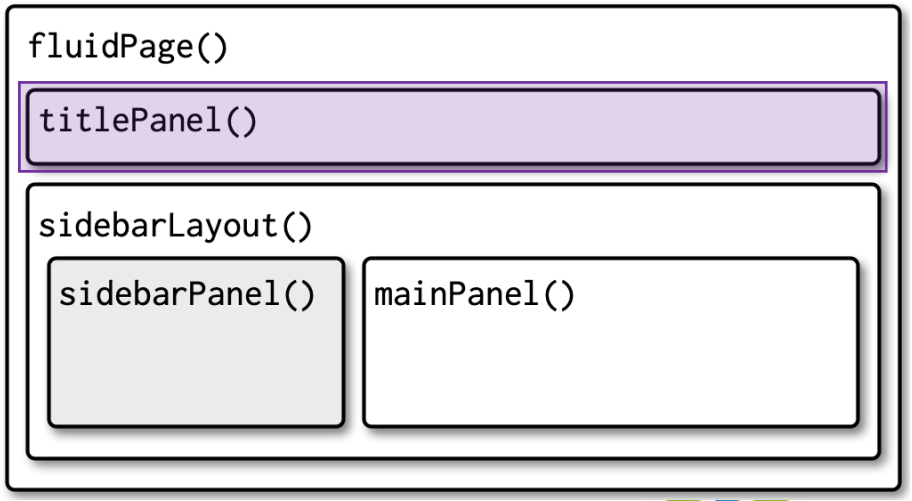
Code Along: titlePanel

- Add title - Remember: it should go inside **fluidPage()**!

- **titlePanel()** creates text formatted as a title at the top of the app.

- **“Run App”** to see what happens!

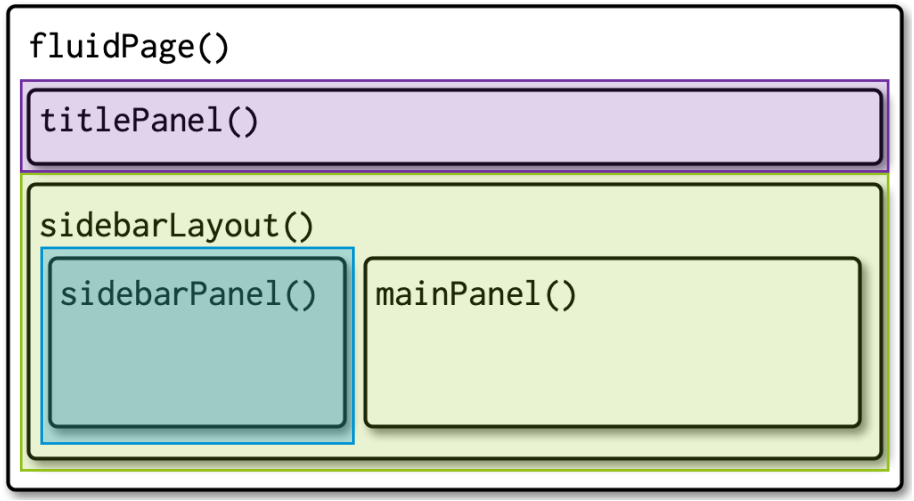
```
library(shiny)

ui <- fluidPage(

  titlePanel("This is a Shiny app")

  ) # end fluidPage

server <- function(input, output){
}

shinyApp(ui = ui, server = server)
```



Code Along: sidebarLayout

- Create a **sidebarLayout()** and add **sidebarPanel()**.
- Add some text to your app.
- Use **br()** for new line.
- **"Run App"** – what happens?

```
library(shiny)

ui <- fluidPage(

  titlePanel("This is a Shiny app"),

  sidebarLayout(
    sidebarPanel(
      "Sidebar",
      br(),
      "Some other text in the sidebar"
    ) # end sidebarPanel
  ) # end sidebarLayout
) # end fluidPage

server <- function(input, output){
}

shinyApp(ui = ui, server = server)
```



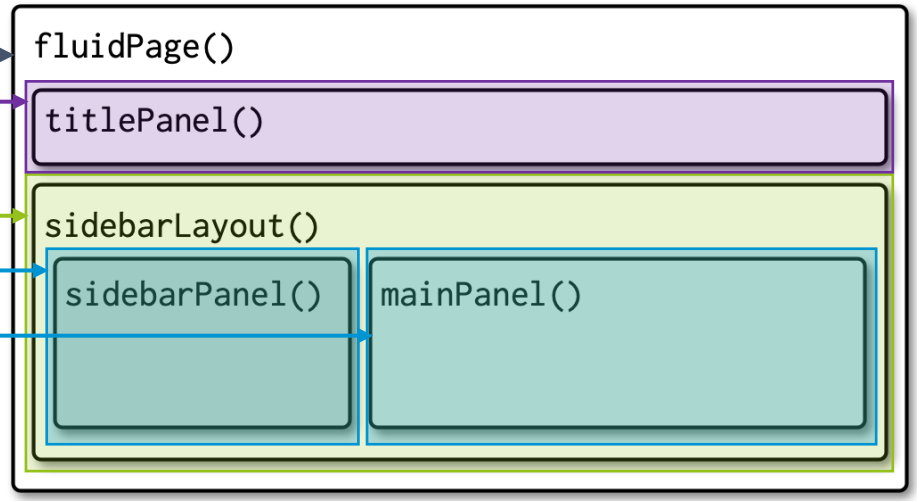
TIPS FOR UI:

- Use Rainbow parentheses
  Tools>Global>Code>Display>
  Tick "Rainbow Parentheses"
- Comment end of functions

Code Along: sidebarLayout

- Add **mainPanel()** and **“Run App”**.

```
library(shiny)

ui <- fluidPage(

  titlePanel("This is a Shiny app"),

  sidebarLayout(
    sidebarPanel(
      "Sidebar",
      br(),
      "Some other text in the sidebar"
      ), # end sidebarPanel

    mainPanel(
      "Main section",
      br(),
      br(),
      br(),
      "Some other text in the main section"
    ) # end mainPanel
    ) # end sidebarLayout
  ) # end fluidPage

server <- function(input, output){
}

shinyApp(ui = ui, server = server)
```



- **NOTE:** sidebarLayout() requires a sidebarPanel() and mainPanel()
  - sidebarPanel would contain user input controls (e.g. radiobuttons and drop-downs)
  - mainPanel would display the output (e.g. plot or table).

Recap

- Shiny apps need two components; UI (controls how the app looks), and server (controls what the app does).

- sidebarPanel() and mainPanel() functions work within sidebarLayout() and fluidPage() to create different sections of the app, and we can change the code in order to change what appears in these sections.

Next

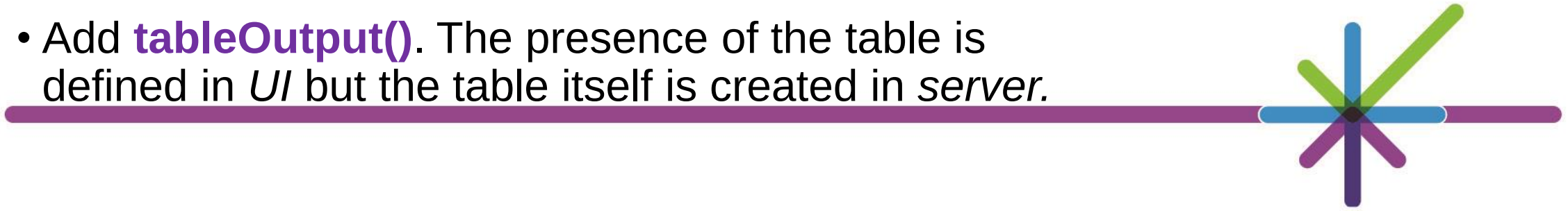
- Using sample data we will build our own multitab app which will include titles, radio buttons, drop-down menus, charts, tables and text.

- We'll be using the nyc_dogs dataset from the NYC

Code Along: Radio buttons

- Clear content of app so you have an empty app with UI and server.
- Load data from data folder.
- Add **radioButtons()** to UI: **radioButtons(inputId, label, choices = NULL)**
  - **inputId** unique ID assigned to widget.
  - **label** names the radioButtons. In this case “Male or Female Dogs?”.
  - **choices** argument is a list of values to select from within the column we have chosen as the input. In this case we list Male and Female. Make sure they match what is in the data.
- Add **tableOutput()**. The presence of the table is defined in *UI* but the table itself is created in *server.*

```
library(shiny)
library(tidyverse)

nyc_dogs <- read_csv("data/nyc_dogs.csv")


ui <- fluidPage(

  radioButtons(inputId = 'gender',
               label = "Male or female dogs?",
               choices = c("Male", "Female")),

  tableOutput("table_output")

) # end fluidPage
```



Code Along: Table output

- Add the table output to the server by creating a table output object: table_output
- Use **renderTable({})** to select dataset, and filter the data based on the user input, in our case, gender: **gender == "input$gender"**.
  - The input$gender should match the radioButtons' inputId in the UI and the output$table_output should match the outputId in the UI
  - Use **slice(1:10)** to only show the first 10 rows of the data.

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

  output$table_output <- renderTable({
    nyc_dogs %>%
      filter(gender == input$gender) %>%
      slice(1:10)
  })

}

# Run the application
shinyApp(ui = ui, server = server)
```

1

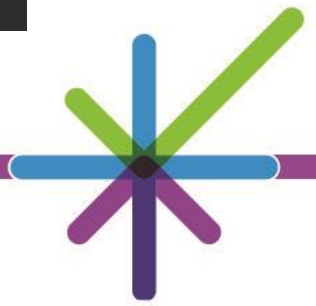
2

UI

```
radioButtons(inputId = 'gender',
             label = "Male or female dogs?",
             choices = c("Male", "Female")),

tableOutput("table_output")
```

2

1

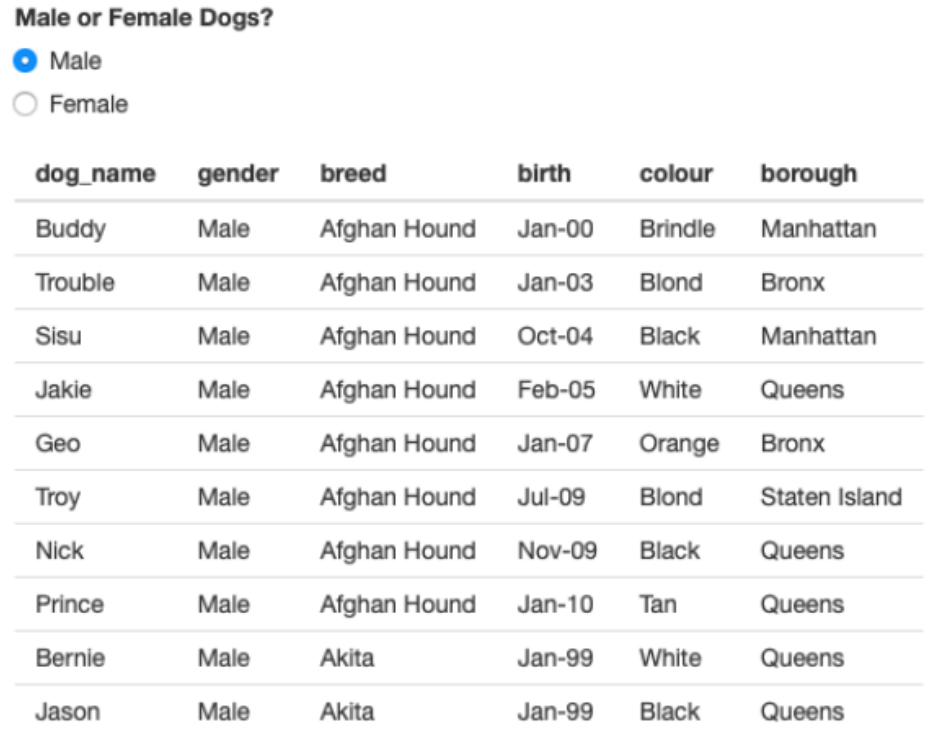


Code Along: Table output

- Running your app will result in a table that looks like this, where you can select the first 10 male or female dogs using the radioButtons.

- We can make the table more interesting using the DT package and the **dataTableOutput()** and **renderDataTable()** functions.

- For more info on the DT package: https://rstudio.github.io/DT/



dog_name	gender	breed	birth	colour	borough
Buddy	Male	Afghan Hound	Jan-00	Brindle	Manhattan
Trouble	Male	Afghan Hound	Jan-03	Blond	Bronx
Sisu	Male	Afghan Hound	Oct-04	Black	Manhattan
Jakie	Male	Afghan Hound	Feb-05	White	Queens
Geo	Male	Afghan Hound	Jan-07	Orange	Bronx
Troy	Male	Afghan Hound	Jul-09	Blond	Staten Island
Nick	Male	Afghan Hound	Nov-09	Black	Queens
Prince	Male	Afghan Hound	Jan-10	Tan	Queens
Bernie	Male	Akita	Jan-99	White	Queens
Jason	Male	Akita	Jan-99	Black	Queens

Code Along: DT table

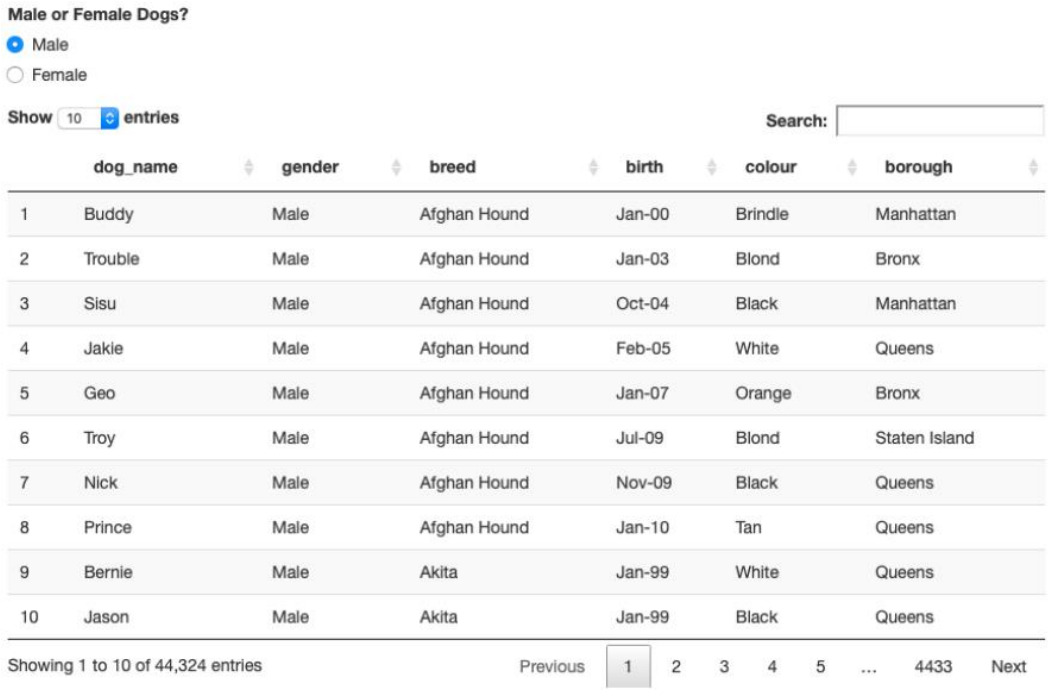
- Our code is very similar.
- **UI**
  - **DT::dataTableOutput("table_output")** instead of **tableOutput("table_output")**
- **Server**
  - **DT::renderDataTable({…})** instead of **renderTable({…})**
- Using DT to sort and search our data also means we no longer need the **slice(1:10)** function in the server.

```
library(shiny)
library(tidyverse)

nyc_dogs <- read_csv("data/nyc_dogs.csv")


ui <- fluidPage(

  radioButtons(inputId = 'gender',
               label = "Male or female dogs?",
               choices = c("Male", "Female")),

  DT::dataTableOutput("table_output")

) # end fluidPage


server <- function(input, output) {

  output$table_output <- DT::renderDataTable({
    nyc_dogs %>%
      filter(gender == input$gender)
  })

}

# Run the application
shinyApp(ui = ui, server = server)
```

Code Along: DT table

- Your app will now look something like this, where you can sort the data in the table and search through it.

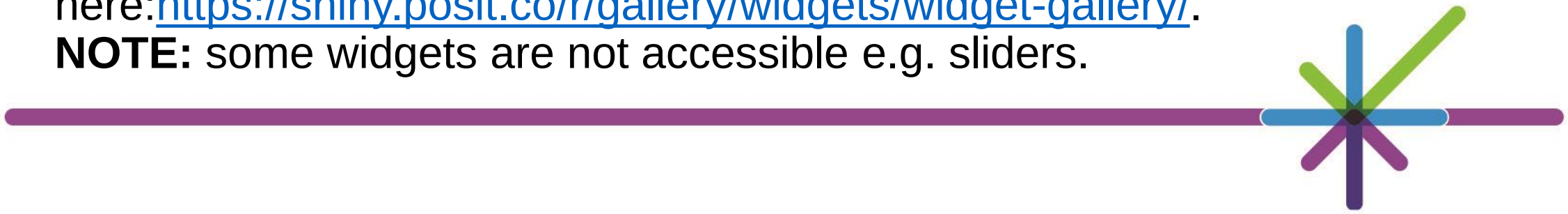
- Other DT features
  - Filter columns
  - Data formatting (e.g. £ and %)
  - Extensive styling options
  - Data download buttons
- We will use the standard table output for the rest of the session.



Code Along: fluidRow() and multiple filters

- **fluidRow()** is another layout option to sidebarLayout() and works with the **column()** function
  - fluidRow: Divides the screen into a horizontal grid
  - column: Divides the fluidRow into columns, max 12

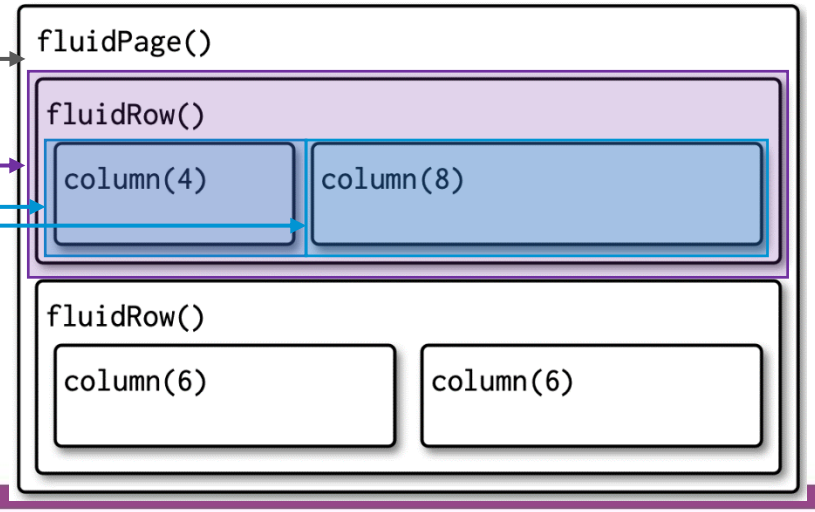
- Drop down filters are created using **selectInput(inputId, label, choices)**
  - **inputId** will be used to access the value.
  - **label** names the drop-down.
  - **choices** argument is a list of values to select from within the column we have chosen as the input.
- Full widget/filter selection can be found here:https://shiny.posit.co/r/gallery/widgets/widget-gallery/.
  NOTE: some widgets are not accessible e.g. sliders.



Code Along: fluidRow() and multiple filters

- Add **fluidRow()**
- Add your **radioButtons()** for gender into a column(3) and a **selectInput()** for breed into a column(3).
  - Get breed choices from nyc_dogs data with **choices = unique(nyc_dogs$breed)**
- **NOTE:** Change back to standard **tableOutput()**
- **"Run App"** and try your filters. What happens?

```
library(shiny)
library(tidyverse)

nyc_dogs <- read_csv("data/nyc_dogs.csv")


ui <- fluidPage(

  fluidRow(
    column(3,
           radioButtons(inputId = 'gender',
                        label = "Male or female dogs?",
                        choices = c("Male", "Female"))
           ), # end column
    column(3,
           selectInput(inputId = "breed",
                       label = "Which breed?",
                       choices = unique(nyc_dogs$breed))
           ) # end column
    ), # end fluidRow

  tableOutput("table_output")

) # end fluidPage
```

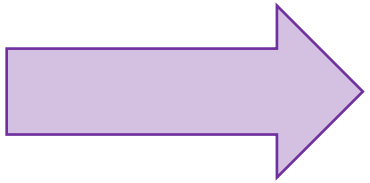


Code Along: fluidRow() and multiple filters

We must add the **server** component for the new breed filter!

- Add a filter for breed in the table_output.
  - Remember to use the correct **inputId** from UI
- **"Run App"**

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

  output$table_output <- renderTable({
    nyc_dogs %>%
      filter(gender == input$gender) %>%
      filter(breed == input$breed)
  })

}

# Run the application
shinyApp(ui = ui, server = server)
```



Male or Female Dogs?

- (•) Male
- ( ) Female

Which Breed? Akita

dog_name	gender	breed	birth	colour	borough
Bernie	Male	Akita	Jan-99	White	Queens
Jason	Male	Akita	Jan-99	Black	Queens
Socrates	Male	Akita	Jan-99	White	Queens
Bear	Male	Akita	Jan-00	Black	Queens
Buster	Male	Akita	Jan-00	White	Queens
Ralph	Male	Akita	Jan-00	Tan	Queens
Rambo	Male	Akita	Jan-00	Tan	Queens
Shoko	Male	Akita	Feb-00	Tan	Brooklyn
Bear	Male	Akita	Jan-01	Blond	Bronx
Darius	Male	Akita	Jan-01	White	Queens

Exercise

- Add two more drop-downs to the app we have which allow you to filter for:
  - Borough, inputId = “borough”
  - Dog colour, inputId = “colour”
- **HINT**: unique inputId, UI + server, max column width is 12

Answer

UI

```
library(shiny)
library(tidyverse)

nyc_dogs <- read_csv("data/nyc_dogs.csv")


ui <- fluidPage(

  fluidRow(
    column(3,
           radioButtons(inputId = 'gender',
                        label = "Male or female dogs?",
                        choices = c("Male", "Female"))
           ), # end column
    column(3,
           selectInput(inputId = "colour",
                       label = "Which colour?",
                       choices = unique(nyc_dogs$colour))
           ), # end column
    column(3,
           selectInput(inputId = "breed",
                       label = "Which breed?",
                       choices = unique(nyc_dogs$breed))
           ), # end column
    column(3,
           selectInput(inputId = "borough",
                       label = "Which borough?",
                       choices = unique(nyc_dogs$borough))
           ), # end column
    ), # end fluidRow

  tableOutput("table_output")

) # end fluidPage
```

Server

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

  output$table_output <- renderTable({
    nyc_dogs %>%
      filter(gender == input$gender) %>%
      filter(colour == input$colour) %>%
      filter(breed == input$breed) %>%
      filter(borough == input$borough)
  })

}

# Run the application
shinyApp(ui = ui, server = server)
```

Next we will add charts!

Code Along: Charts

- Plots are displayed using **plotOutput()** in the UI
- In server, instead of **renderTable({})**, we use **renderPlot({})**
- Plots are created using ggplot() and/or plotly() (a package for adding interactivity to charts).
- You can create the plots in Plotly directly or pass a ggplot object to ggplotly(), turning your ggplot into an interactive plotly object. See here for more info on plotly: https://plotly.com/r/

Code Along: Charts UI

- Next, we will add two bar charts; one for colour and one for borough.

- Keep the radio buttons for selecting gender, and the drop-down menu for selecting breed. Comment out the other filters.

- Add new **fluidRow()** with two **columns()** and add one chart in each; “colour_barchart” and “borough_barchart”.
    - Remember to use **plotOutput()**

```
library(shiny)
library(tidyverse)

nyc_dogs <- read_csv("data/nyc_dogs.csv")

ui <- fluidPage(

  fluidRow(
    column(6,
           radioButtons(inputId = 'gender',
                        label = "Male or female dogs?",
                        choices = c("Male", "Female"))
           ), # end column
    column(6,
           selectInput(inputId = "breed",
                       label = "Which breed?",
                       choices = unique(nyc_dogs$breed))
           ), # end column
    ), # end fluidRow

  fluidRow(
    column(6,
           plotOutput(outputId = "colour_barchart")
           ), # end column
    column(6,
           plotOutput(outputId = "borough_barchart")
    ), # end column
  ) # end fluidRow
) # end fluidPage
```

Code Along: Charts Server

- Create a dataset called filtered_data that filters based on user input for gender and breed
  - Creating a separate dataset prevents us from repeating the data code for each plot
- Create two bar charts using **ggplot()**; colour_barchart and borough_barchart. The names must match the **outputId** in the UI
- Remember to use **renderPlot({})**
- **"Run App"** – what happens?

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

  filtered_data <- nyc_dogs %>%
    filter(gender == input$gender) %>%
    filter(breed == input$breed)

  output$colour_barchart <- renderPlot({
    ggplot(filtered_data) +
      geom_bar(aes(x = colour))
  })

  output$borough_barchart <- renderPlot({
    ggplot(filtered_data) +
      geom_bar(aes(x = borough))
  })

}

# Run the application
shinyApp(ui = ui, server = server)
```

Code along: Reactivity

When we run the previous code, we get an error.

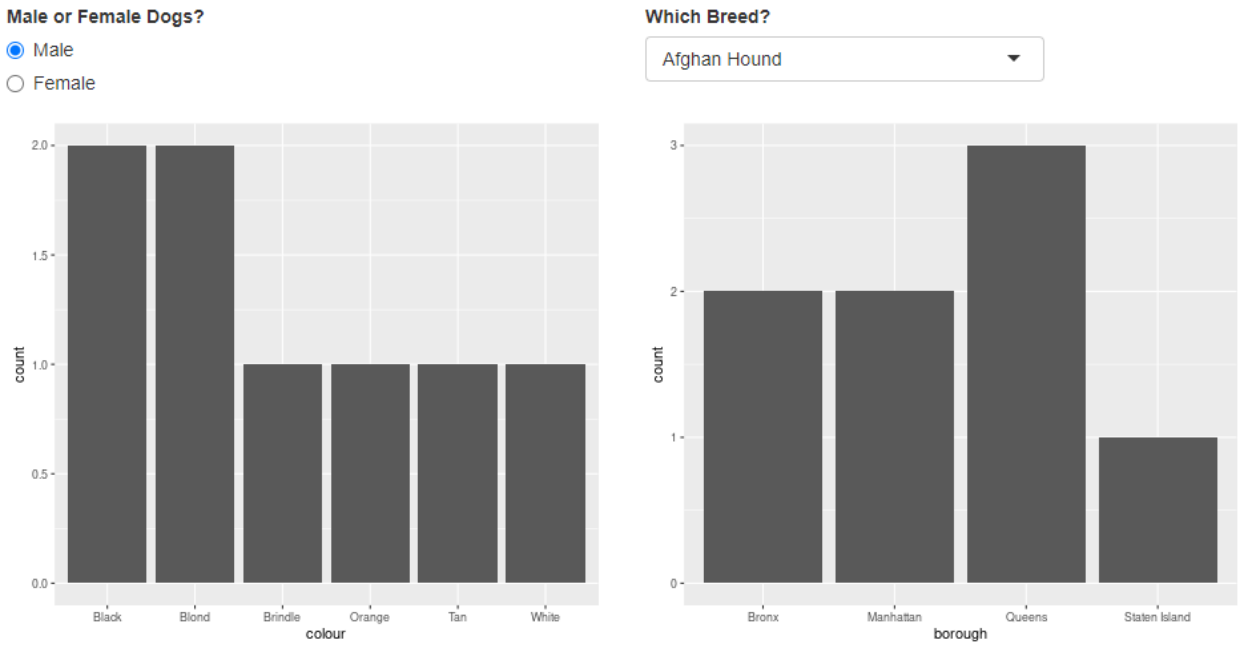
```
Do you need to wrap inside reactive() or observer()?
```

- We've used regular R code to filter our data, but the filter for this object needs to be reactive depending on what the user inputs.
- A reactive object can change during the running of the app, in our case, filtered_data.
- Wrap filtered_data in the **reactive({})** function.
  - This defines the data as having a reactive value and must therefore be followed by brackets when used; **filtered_data()**
- **"Run App"**

```
# Run the application
shinyApp(ui = ui, server = server)


server <- function(input, output) {

  filtered_data <- reactive({
    nyc_dogs %>%
      filter(gender == input$gender) %>%
      filter(breed == input$breed)
  })

  output$colour_barchart <- renderPlot({
    ggplot(filtered_data()) +
      geom_bar(aes(x = colour))
  })

  output$borough_barchart <- renderPlot({
    ggplot(filtered_data()) +
      geom_bar(aes(x = borough))
  })

}

# Run the application
shinyApp(ui = ui, server = server)
```

Code Along: Charts

Our app should now look like this, where the bar charts are reactive to selections made by the user either by the radioButton or the drop-down menu.



Recap

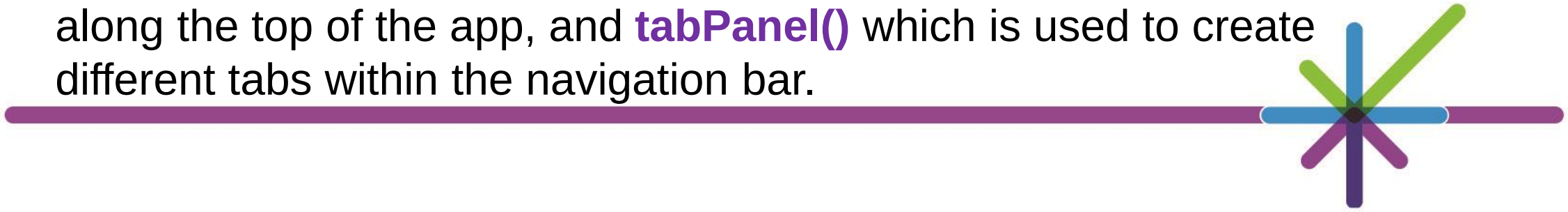
- We have created an app which displays our NYC dogs data in a table which can be filtered using radioButtons and drop-down menus.

- We have also created an app which displays our NYC dogs data in bar charts which are reactive to radioButtons and drop-down menus.

Next

- A multi-tab dashboard where we can display both tables and charts

- An introduction to **navbarPage()** which creates a navigation bar along the top of the app, and **tabPanel()** which is used to create different tabs within the navigation bar.



Including multiple tabs

- Multi-tab dashboards require a lot of nesting of functions within other functions.
  - Having Rainbow Brackets activated is really helpful (compulsory) for this!

- To have, for example, charts in one tab and the table in another, we need to wrap each code block (one for charts and one for tables) inside its own **tabPanel()** function.

- These **tabPanel()s** then need to be nested inside the **navbarPage()** function which creates the overarching navigation bar along the top of the app.

Code Along: Multiple tabs – Tab 1 (Table) UI

- Create and name a **navbarPage()** with a nested and named **tabPanel(title = “”)**.
- Add your **fluidRow()** with your four filters (uncomment the borough and breed drop-downs) to the **tabPanel()**.
- Only keep the **tableOutput(“table_output”)** – remember to keep it in the **tabPanel()**

```
library(shiny)
library(tidyverse)

nyc_dogs <- read_csv("data/nyc_dogs.csv")

ui <- fluidPage(

  titlePanel("NYC DOGS"),

  navbarPage("Navigation Bar",
             tabPanel(title = "Table",
                      fluidRow(
                        column(3,
                               radioButtons(inputId = 'gender',
                                            label = "Male or female dogs?",
                                            choices = c("Male", "Female"))
                               ), # end column
                        column(3,
                               selectInput(inputId = "colour",
                                           label = "Which colour?",
                                           choices = unique(nyc_dogs$colour))
                               ), # end column
                        column(3,
                               selectInput(inputId = "borough",
                                           label = "Which borough?",
                                           choices = unique(nyc_dogs$borough))
                               ), # end column
                        column(3,
                               selectInput(inputId = "breed",
                                           label = "Which breed?",
                                           choices = unique(nyc_dogs$breed))
                               ) # end column
                        ), # end fluidRow

                      tableOutput("table_output")), # end tabPanel
  ) # end navbarPage
) # end fluidPage
```

Code Along: Multiple tabs – Tab 1 (Table) Server

- Create a new reactive dataset, **table_data**
- Remember to add your four filter inputs
- **“Run App”**

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

  table_data <- reactive({
    nyc_dogs %>%
      filter(gender == input$gender) %>%
      filter(breed ==  input$breed) %>%
      filter(colour ==  input$colour) %>%
      filter(borough ==  input$borough)
  })

  output$table_output <- renderTable({
    table_data()
  })
}
```

Code Along: Multiple tabs – Tab 2 (Chart) UI

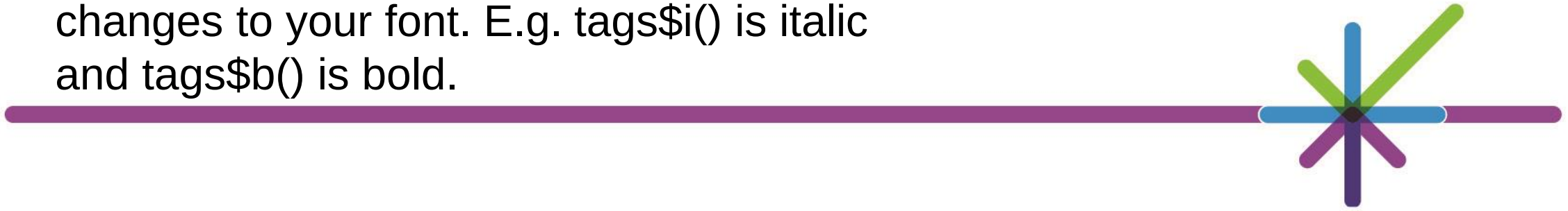
- Create and name a second **tabPanel()**.

- Add your **fluidRow()** with two new filters; gender and breed. They need new **inputIds** as every filter needs a unique id.

- Add another **fluidRow()** with the two bar charts we made earlier.

- **NOTE:** You can use tags to make changes to your font. E.g. tags$i() is italic and tags$b() is bold.

```
tabPanel(title = "Plot",
         fluidRow(
           column(6,
                  radioButtons(inputId = "gender_chart",
                               label = tags$i("Male or Female Dogs?"),
                               choices = c("Male", "Female"))
           ), # end column
           column(6,
                  selectInput(inputId = "breed_chart",
                              label = tags$i("Which Breed?"),
                              choices = unique(nyc_dogs$breed))
           ) # end column
         ), # end fluidRow

         fluidRow(
           column(6,
                  plotOutput("colour_barchart")
           ), # end column
           column(6,
                  plotOutput("borough_barchart"))
           ) # end fluidRow
         ), # end tabPanel
```



Code Along: Multiple tabs – Tab 2 (Chart) Server

- Create a new reactive dataset, **plot_data**
- Remember to add your two filter inputs with the new **inputIds**
- Add the two bar charts you made before, remember to use your new reactive dataset
- **"Run App"**

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

  table_data <- reactive({
    nyc_dogs %>%
      filter(gender == input$gender) %>%
      filter(breed ==  input$breed) %>%
      filter(colour ==  input$colour) %>%
      filter(borough ==  input$borough)
  })

  output$table_output <- renderTable({
    table_data()
  })

  plot_data <- reactive({
    nyc_dogs %>%
      filter(gender == input$gender_chart) %>%
      filter(breed ==  input$breed_chart)
  })

  output$colour_barchart <- renderPlot({
    ggplot(plot_data()) +
      geom_bar(aes(x = colour))
  })

  output$borough_barchart <- renderPlot({
    ggplot(plot_data()) +
      geom_bar(aes(x = borough))
  })

}
```

Code Along: Multiple tabs – Tab 3 (Info) UI

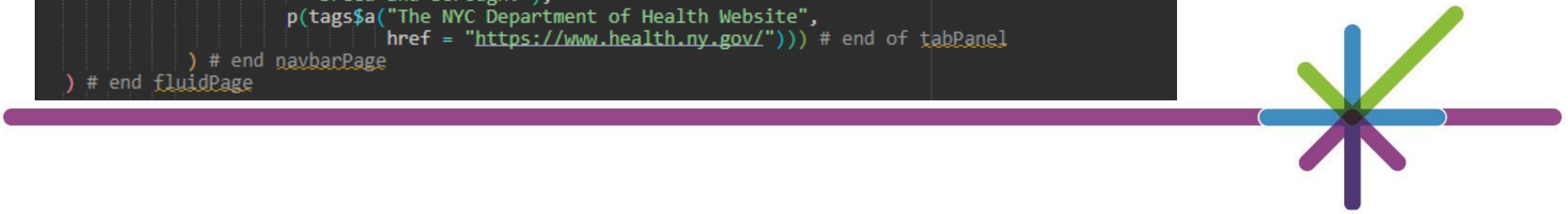
- Create and name a third **tabPanel()**.

- Add text to your new tabPanel using **p()**.
  - p() indicates a paragraph

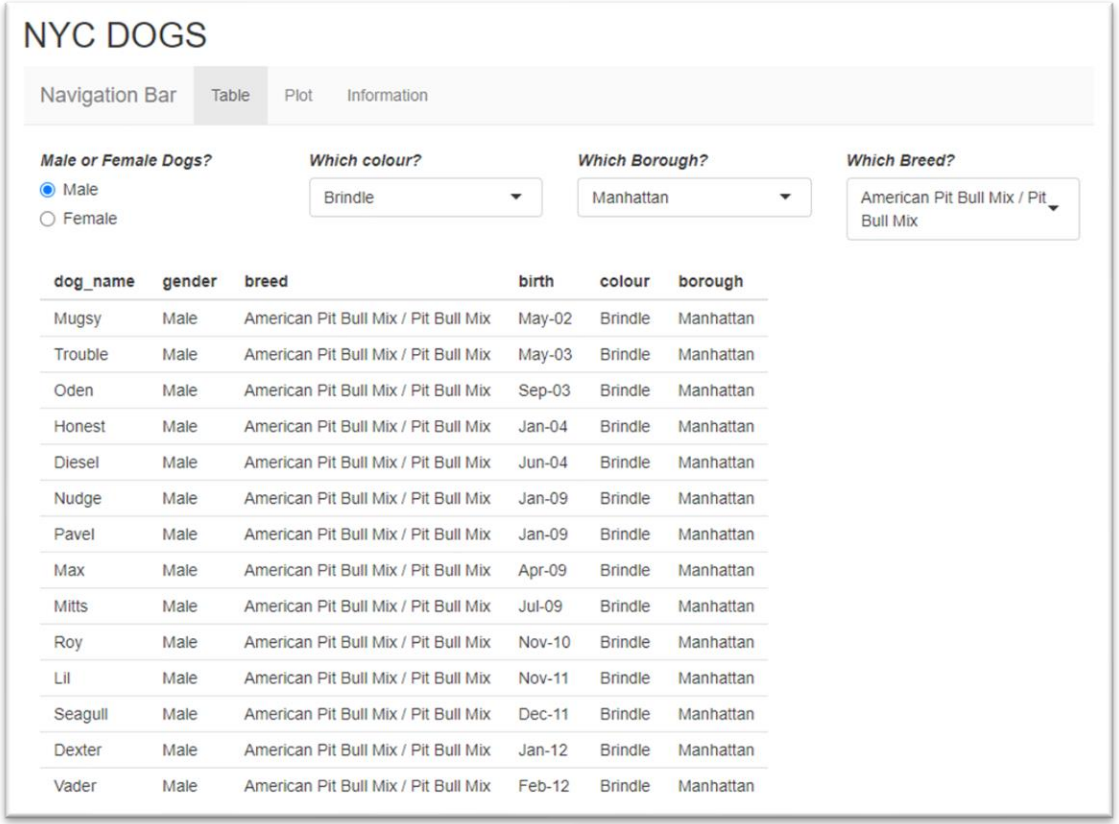
- Use tags$a to create a link to the NYC Department of Health website.

- No need to update server as we only add text.

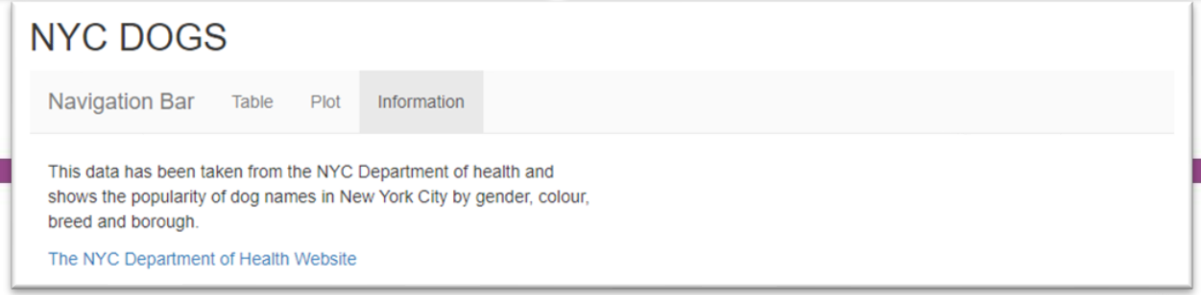
```
                    ), # end tabPanel
            tabPanel(title = "Information",
                     p("This data has been taken from the NYC Department of health and", br(),
                       "shows the popularity of dog names in New York City by gender, colour,", br(),
                       "breed and borough."),
                     p(tags$a("The NYC Department of Health Website",
                              href = "https://www.health.ny.gov/"))) # end of tabPanel
            ) # end navbarPage
) # end fluidPage
```



Code Along: Final app



dog_name	gender	breed	birth	colour	borough
Mugsy	Male	American Pit Bull Mix / Pit Bull Mix	May-02	Brindle	Manhattan
Trouble	Male	American Pit Bull Mix / Pit Bull Mix	May-03	Brindle	Manhattan
Oden	Male	American Pit Bull Mix / Pit Bull Mix	Sep-03	Brindle	Manhattan
Honest	Male	American Pit Bull Mix / Pit Bull Mix	Jan-04	Brindle	Manhattan
Diesel	Male	American Pit Bull Mix / Pit Bull Mix	Jun-04	Brindle	Manhattan
Nudge	Male	American Pit Bull Mix / Pit Bull Mix	Jan-09	Brindle	Manhattan
Pavel	Male	American Pit Bull Mix / Pit Bull Mix	Jan-09	Brindle	Manhattan
Max	Male	American Pit Bull Mix / Pit Bull Mix	Apr-09	Brindle	Manhattan
Mitts	Male	American Pit Bull Mix / Pit Bull Mix	Jul-09	Brindle	Manhattan
Roy	Male	American Pit Bull Mix / Pit Bull Mix	Nov-10	Brindle	Manhattan
Lil	Male	American Pit Bull Mix / Pit Bull Mix	Nov-11	Brindle	Manhattan
Seagull	Male	American Pit Bull Mix / Pit Bull Mix	Dec-11	Brindle	Manhattan
Dexter	Male	American Pit Bull Mix / Pit Bull Mix	Jan-12	Brindle	Manhattan
Vader	Male	American Pit Bull Mix / Pit Bull Mix	Feb-12	Brindle	Manhattan



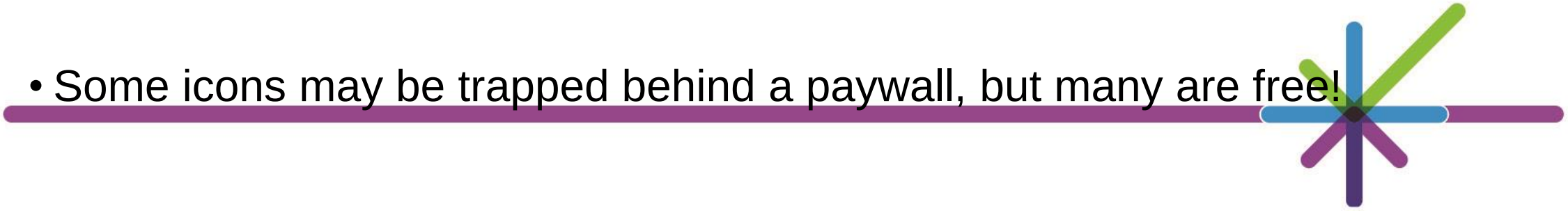
Shiny themes and icons

- We can use basic HTML and CSS code to set fonts and themes for our dashboard and insert icons in the navigation bar for each section.

- Shiny has a variety of ready made themes that can be used in your dashboard, try out the theme selector: https://shiny.rstudio.com/gallery/shiny-theme-selector.html

- There are also a variety of icons that can be added to your Shiny dashboard using Font Awesome and Glyphicon: https://fontawesome.com/v5.15/icons?d=gallery&p=2 https://getbootstrap.com/docs/3.3/components/#glyphicons

- Some icons may be trapped behind a paywall, but many are free!



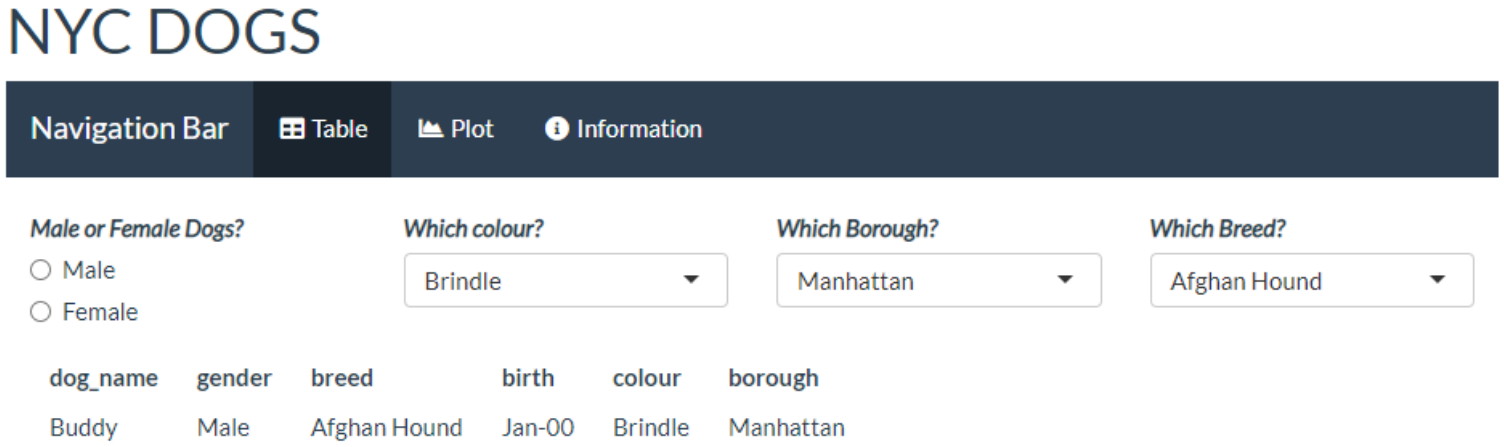
Shiny themes and icons

- To use themes and icons, we need to load two more packages:
  - library(shinythemes)
  - library(shinycssloaders)
- To add a theme, add **theme = shinytheme("*name of theme*")** in **fluidPage()**
- To add an icon, add **icon = icon("*name of icon*")** to your **tabPanel()**
- **NOTE:** You can wrap your NYC DOGS title in tags$h1() to set text as HEADER 1. Adding html tags aids screen readers.
- **EXERCISE:** Add icons to all tabs. See previous slide for link to Font Awesome and Glyphicon

```
ui <- fluidPage(

  theme = shinytheme("flatly"),

  titlePanel(tags$h1("NYC DOGS")),

  navbarPage("Navigation Bar",
             tabPanel(title = "Table",
                      icon = icon("table"),
                      fluidRow(
                        column(3,
                               radioButtons(inputId = 'gender',
                                            label = "Male or female dogs?",
                                            choices = c("Male", "Female"))
                               ), # end column
                        column(3,
                               selectInput(inputId = "colour",
                                           label = "Which colour?",
                                           choices = unique(nyc_dogs$colour))
                               ), # end column
                        column(3,
                               selectInput(inputId = "borough",
                                           label = "Which borough?",
                                           choices = unique(nyc_dogs$borough))
                               ), # end column
                        column(3,
                               selectInput(inputId = "breed",
                                           label = "Which breed?",
                                           choices = unique(nyc_dogs$breed))
                               ) # end column
                        ), # end fluidRow
```

Exercise: Shiny themes and icons

- Now our Shiny app looks a bit more interesting!



- Use the theme and icons links to make your dashboard look nice!
- If you are confident with ggplot2, make your plots look nicer or if you want a challenge, make them in plotly and play around with interactivity.

Practice makes perfect

- You have created a Shiny dashboard from scratch which contains tabulated data, charts and information text tabs.
- You have made the data reactive to whatever the user inputs and you have used different themes, icons and text formatting to make it stand out.

- If you have time before our next session, try playing around with this dashboard.
  - Swap your fluidRow() and column() grid layouts for something else such as sidebarLayout(), sidebarPanel() and mainPanel() to see how the layout changes.
  - Add new tabs with different charts.
  - Attempt to make a similar dashboard for a whole new dataset. [rfordatascience/tidytuesday: Official repo for the #tidytuesday project (github.com)]

Next time

- Splitting big dashboards: UI, Server and Global scripts
- Using multiple Public Health Scotland datasets
- Use of specific Public Health Scotland colours and logos
- Modals and help buttons
- Data downloads
- The importance of using GitHub

End of day 1 – any questions?