Plotting 2.0: Shiny

TheRBootcamp, http://therbootcamp.github.io (http://therbootcamp.github.io)

Source: https://www.rstudio.com/ (https://www.rstudio.com/)

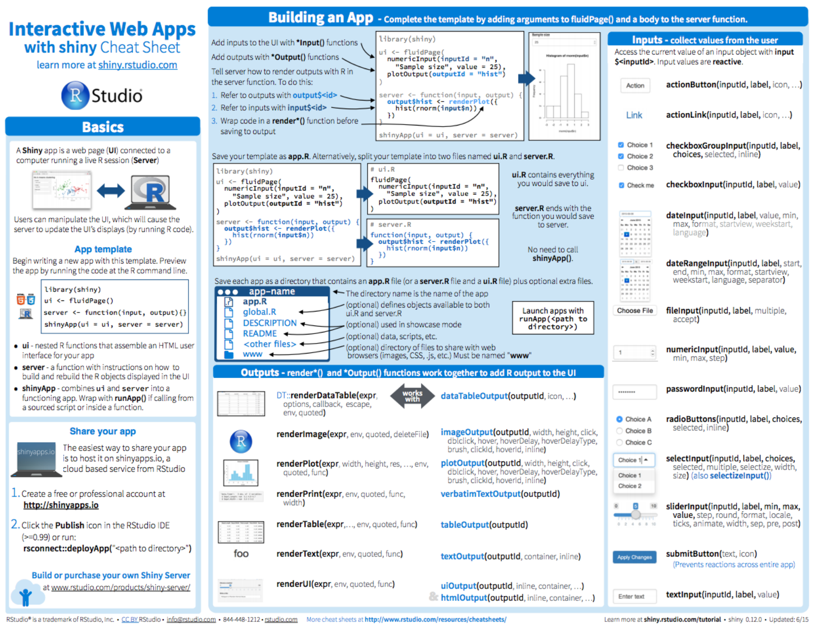
Slides

Here are the introduction slides for this practical on Plotting 2.0: Shiny! (https://therbootcamp.github.io/_sessions/D3S3_PlottingII/PlottingII.html)

Overview

In this practical you'll practice creating interactive graphics with `shiny`.

Cheatsheet



Interactive Web Apps with shiny Cheat Sheet
learn more at shiny.rstudio.com

RStudio

Basics

A **Shiny** app is a web page (**UI**) connected to a computer running a live R session (**Server**)

Users can manipulate the UI, which will cause the server to update the UI's displays (by running R code).

App template

Begin writing a new app with this template. Preview the app by running the code at the R command line.

```
library(shiny)
ui <- fluidPage()
server <- function(input, output){}
shinyApp(ui = ui, server = server)
```

- **ui** - nested R functions that assemble an HTML user interface for your app
- **server** - a function with instructions on how to build and rebuild the R objects displayed in the UI
- **shinyApp** - combines `ui` and `server` into a functioning app. Wrap with **runApp()** if calling from a sourced script or inside a function.

Share your app

The easiest way to share your app is to host it on shinyapps.io, a cloud based service from RStudio

1. Create a free or professional account at **http://shinyapps.io**
2. Click the **Publish** icon in the RStudio IDE (>=0.99) or run: **rsconnect::deployApp(**"<path to directory>"**)**

Build or purchase your own Shiny Server at www.rstudio.com/products/shiny-server/

Building an App - Complete the template by adding arguments to fluidPage() and a body to the server function.

Add inputs to the UI with ***Input()** functions

Add outputs with ***Output()** functions

Tell server how to render outputs with R in the server function. To do this:

1. Refer to outputs with **output$<id>**
2. Refer to inputs with **input$<id>**
3. Wrap code in a **render*()** function before saving to output

```
library(shiny)

ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}

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

Save your template as **app.R**. Alternatively, split your template into two files named **ui.R** and **server.R**.

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```

```
# ui.R
fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
```

```
# server.R
function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
```

ui.R contains everything you would save to ui.

server.R ends with the function you would save to server.

No need to call **shinyApp()**.

Save each app as a directory that contains an **app.R** file (or a **server.R** file and a **ui.R** file) plus optional extra files.

app-name	
	The directory name is the name of the app
app.R	
global.R	(optional) defines objects available to both ui.R and server.R
DESCRIPTION	(optional) used in showcase mode
README	(optional) used in showcase mode
<other files>	(optional) data, scripts, etc.
www	(optional) directory of files to share with web browsers (images, CSS, .js, etc.) Must be named "**www**"

Launch apps with `runApp(<path to directory>)`

Outputs - render*() and *Output() functions work together to add R output to the UI

render*()	works with	*Output()
DT::**renderDataTable(expr**, options, callback, escape, env, quoted**)**		**dataTableOutput(outputId**, icon, ...**)**
renderImage(expr, env, quoted, deleteFile**)**		**imageOutput(outputId**, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline**)**
renderPlot(expr, width, height, res, ..., env, quoted, func**)**		**plotOutput(outputId**, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline**)**
renderPrint(expr, env, quoted, func, width**)**		**verbatimTextOutput(outputId)**
renderTable(expr,..., env, quoted, func**)**		**tableOutput(outputId)**
renderText(expr, env, quoted, func**)**		**textOutput(outputId**, container, inline**)**
renderUI(expr, env, quoted, func**)**		**uiOutput(outputId**, inline, container, ...**)** & **htmlOutput(outputId**, inline, container, ...**)**

Inputs - collect values from the user

Access the current value of an input object with **input$<inputId>**. Input values are **reactive**.

Action	**actionButton(inputId, label,** icon, ...**)**
Link	**actionLink(inputId, label,** icon, ...**)**
Choice 1 Choice 2 Choice 3	**checkboxGroupInput(inputId, label, choices,** selected, inline**)**
Check me	**checkboxInput(inputId, label,** value**)**
	dateInput(inputId, label, value, min, max, format, startview, weekstart, language**)**
	dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator**)**
Choose File	**fileInput(inputId, label,** multiple, accept**)**
1	**numericInput(inputId, label, value,** min, max, step**)**
••••••••	**passwordInput(inputId, label,** value**)**
Choice A Choice B Choice C	**radioButtons(inputId, label, choices,** selected, inline**)**
Choice 1 Choice 1 Choice 2	**selectInput(inputId, label, choices,** selected, multiple, selectize, width, size**)** (also **selectizeInput()**)
0 5 10	**sliderInput(inputId, label, min, max, value,** step, round, format, locale, ticks, animate, width, sep, pre, post**)**
Apply Changes	**submitButton(**text, icon**)** (Prevents reactions across entire app)
Enter text	**textInput(inputId, label,** value**)**

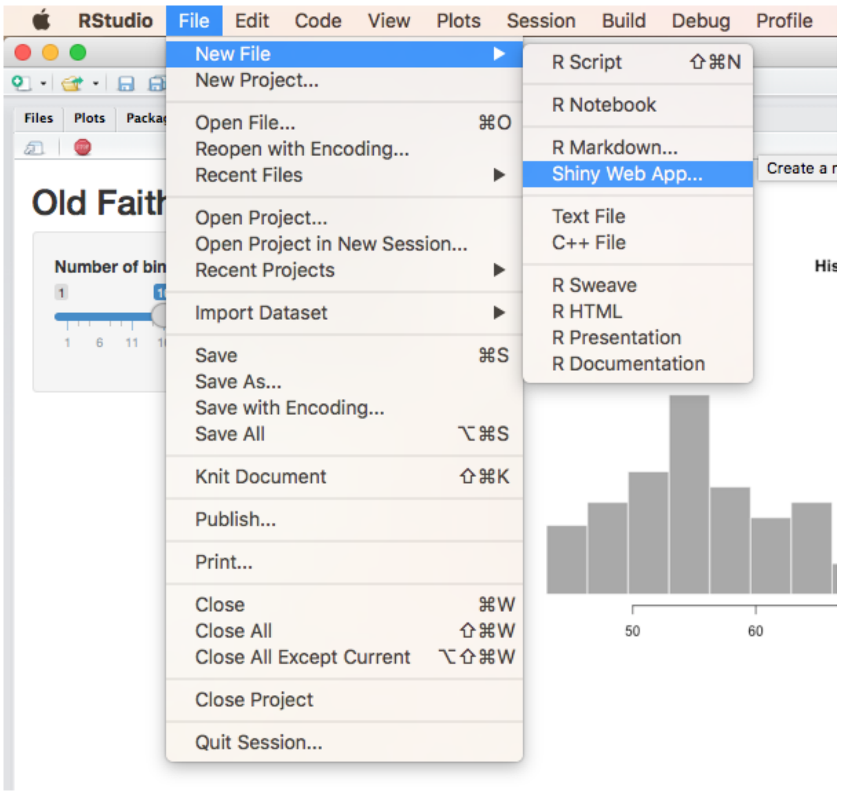
RStudio® is a trademark of RStudio, Inc. • CC BY RStudio • info@rstudio.com • 844-448-1212 • rstudio.com   More cheat sheets at http://www.rstudio.com/resources/cheatsheets/   Learn more at **shiny.rstudio.com/tutorial** • shiny 0.12.0 • Updated: 6/15

http://shiny.rstudio.com/ (http://shiny.rstudio.com/)../_image/shiny-cheatsheet.pdf

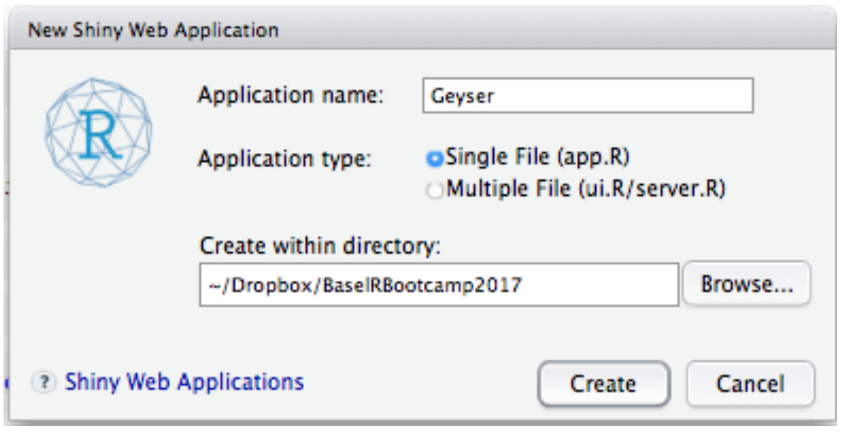
If you don't have it already, you can access the `shiny` cheatsheet here http://shiny.rstudio.com/images/shiny-cheatsheet.pdf (http://shiny.rstudio.com/images/shiny-cheatsheet.pdf)

Examples

- Because Shiny code looks quite a bit different from standard R code, the best way to look at examples is to see a new app in action. In RStudio, click File – New File – Shiny Web App



- Save the app under the name “geyser” in a directory of your choosing



New Shiny Web Application

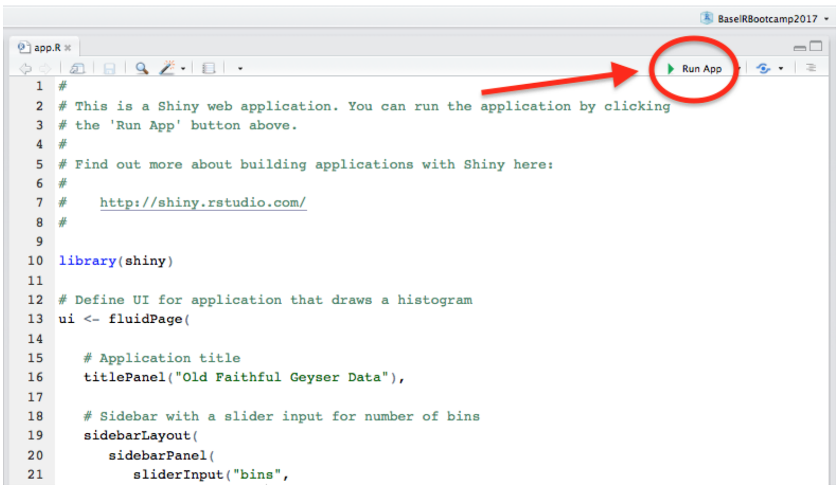
Application name: Geyser

Application type: [✓ Single File (app.R)] Multiple File (ui.R/server.R)

Create within directory: ~/Dropbox/BaselRBootcamp2017 Browse...

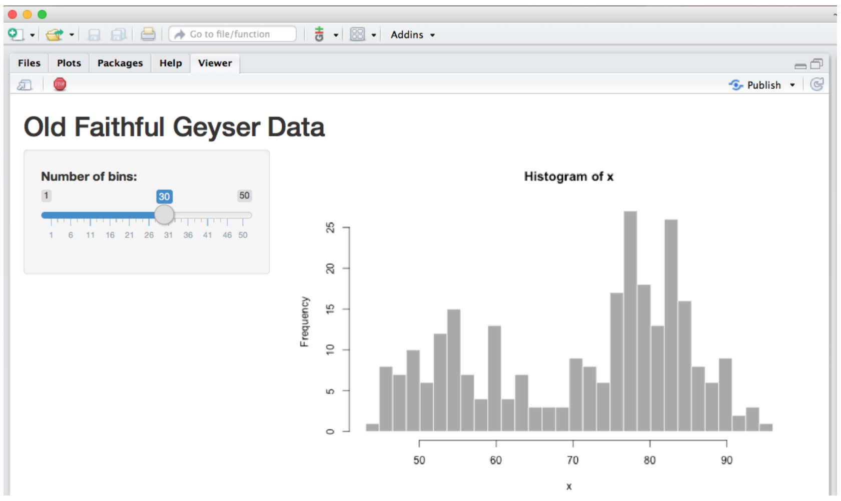
Shiny Web Applications Create Cancel

- A new file called `geyser.app` should now be open. Click the “Run App” button to open the app.



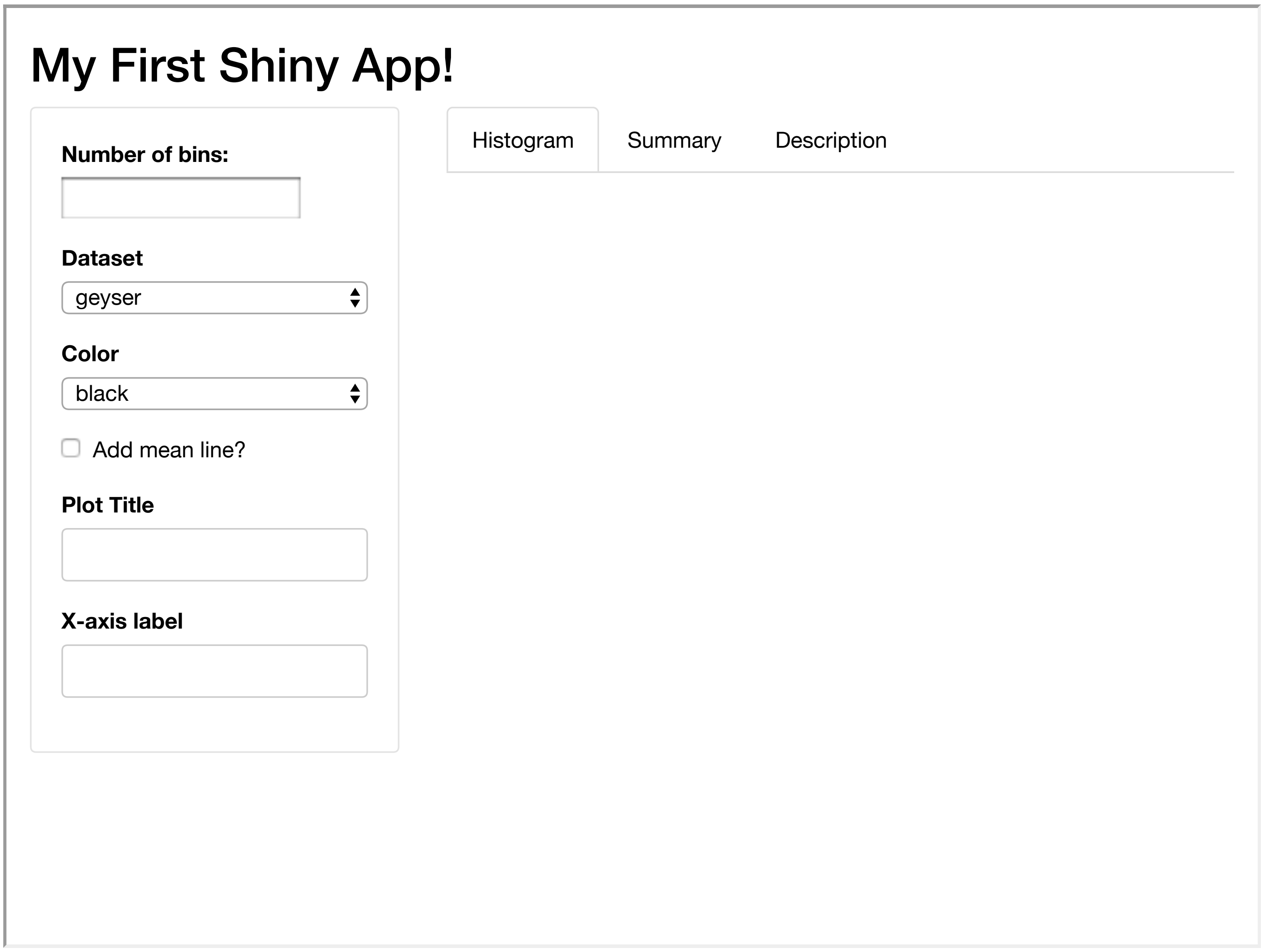
```
#
# This is a Shiny web application. You can run the application by clicking
# the 'Run App' button above.
#
# Find out more about building applications with Shiny here:
#
#    http://shiny.rstudio.com/
#

library(shiny)

# Define UI for application that draws a histogram
ui <- fluidPage(

   # Application title
   titlePanel("Old Faithful Geyser Data"),

   # Sidebar with a slider input for number of bins
   sidebarLayout(
      sidebarPanel(
         sliderInput("bins",
```

- Your app should now be up in a new window! Play around with the Number of Bins slider to see its effect on the plot



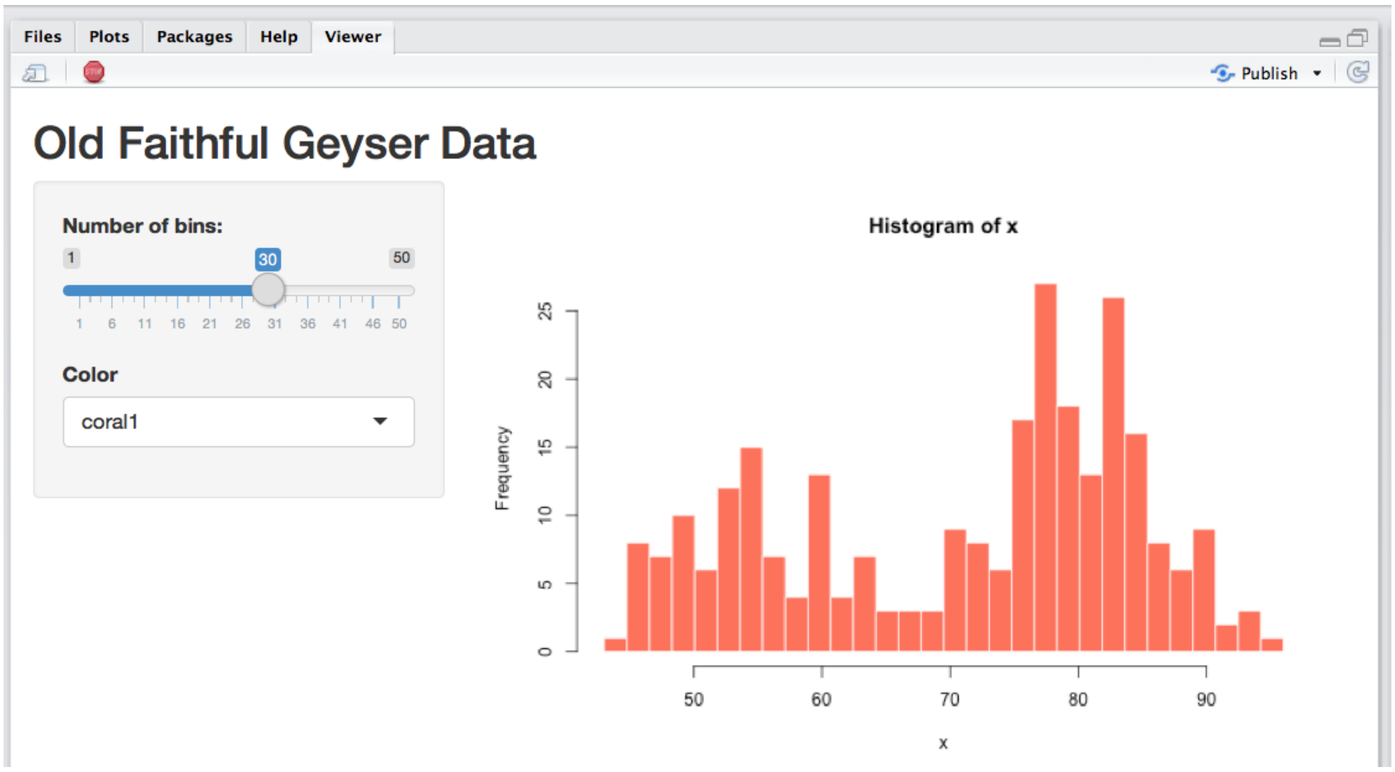
Part A: Get started

By the end of this practical, you'll try to create the following app:



1. Start by following the example above so you have your own `geyser.app`
2. Explore the code in the app to see how it works. Where in the app is the user interface defined? Where is the server code? Where does the server code get access to the vector of geyser data? In what object is the geyser data stored?
3. Change the main title of the app from "Old Faithful Geyser Data" to "___'s first Shiny App!" using your first name.
4. In the server code, change the color of the bins of the histogram to `"red"` . Reload your app to see your changes!

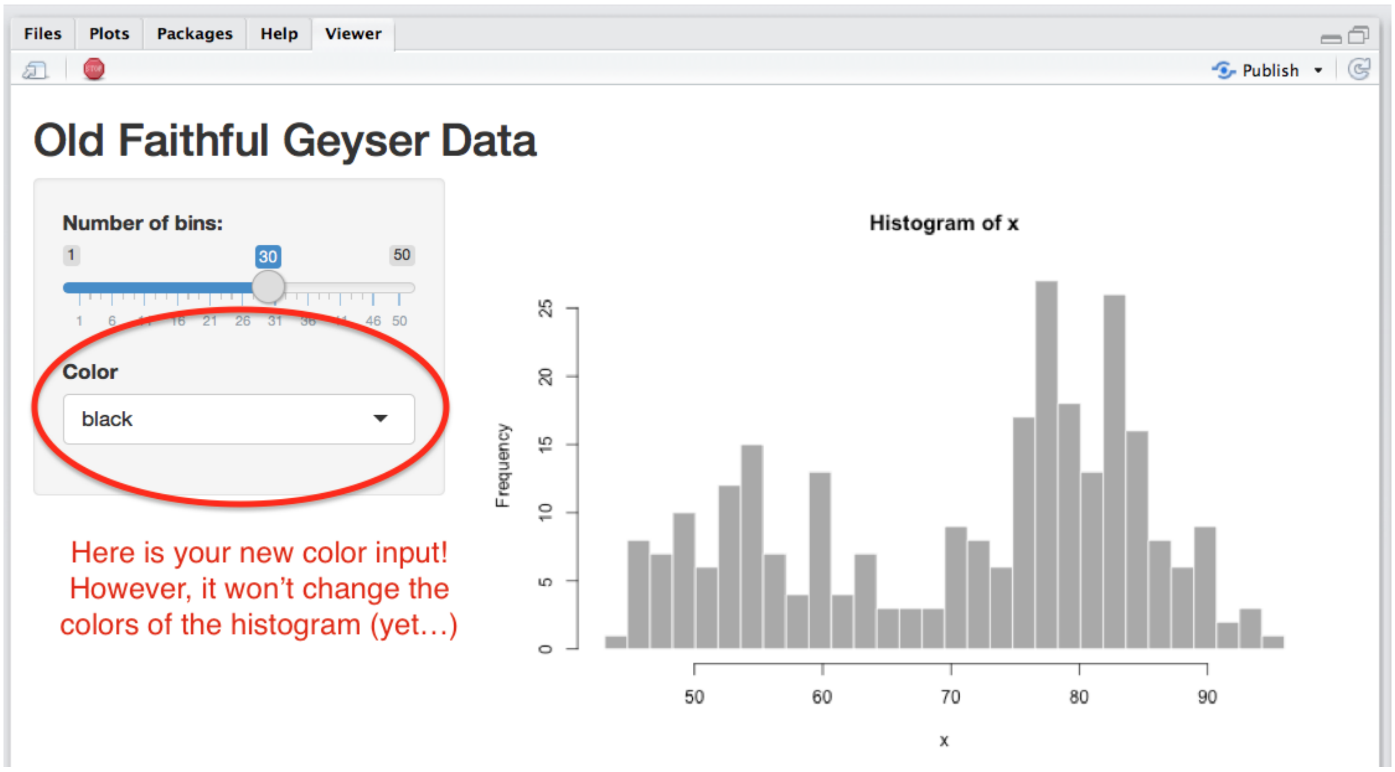
Part B: Add a `selectInput` to change bin colors



5. Now let’s add a new widget that allows the user to control the color of the histogram as a Select Box. Below the bin slider input, add a new `selectInput` like in the code below.

```
# Add a new selectInput to your app below the sliderInput():

 sliderInput("bins",
             "Number of bins:",
             min = 1,
             max = 50,
             value = 30),    # <- Dont forget to put a comma here

 # HERE!!! Add Bin Color Select Input
 selectInput(inputId = "bincolor",
             label = "Color",
             choices = c("red", "blue", "green", "black"),
             selected = "black")
```

6. Now reload your app. You should now see a new select box on the sidebar panel. Try playing around with it by changing the color to green. Nothing happens!??! Why is that???



Old Faithful Geyser Data

Number of bins: 30

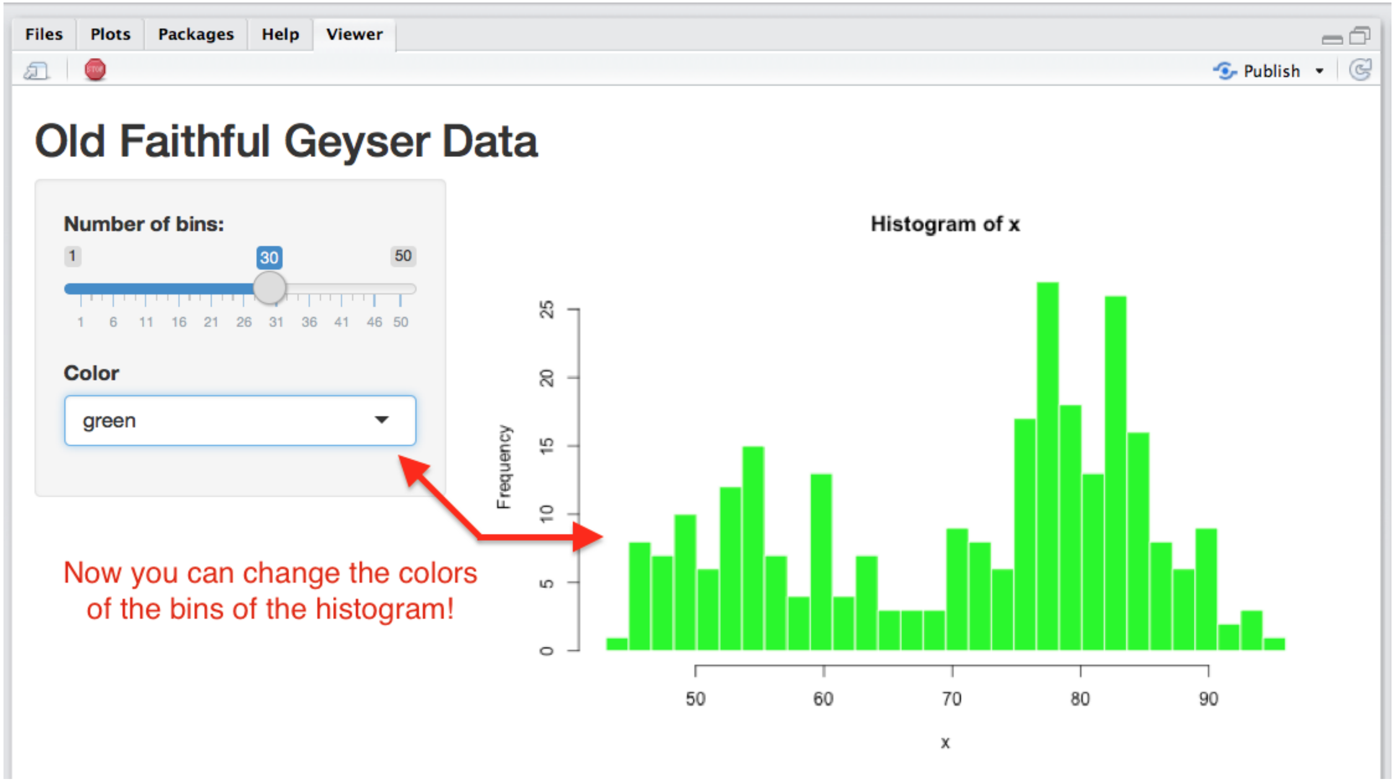
Color: black

Here is your new color input! However, it won't change the colors of the histogram (yet…)

7. The problem is that we changed the input, but we haven't told the actual plotting code in the server portion of the app (specifically `hist()`) to use the new color input! In the server portion of the app, tell the histogram to use the new color input as follows:

```
# Make this change to the hist() function in your app
#   This will tell Shiny that the colors of the histogram
#   bins should change based on the user's input

hist(x,
     breaks = bins,
     col = input$bincolor,    # HERE!!! Change color based on input$bincolor
     border = 'white')
```

8. Now reload the app and try changing the histogram bin color using the select input. You should now see a change in the color!



Old Faithful Geyser Data

Number of bins: 30

Color: green

Now you can change the colors of the bins of the histogram!

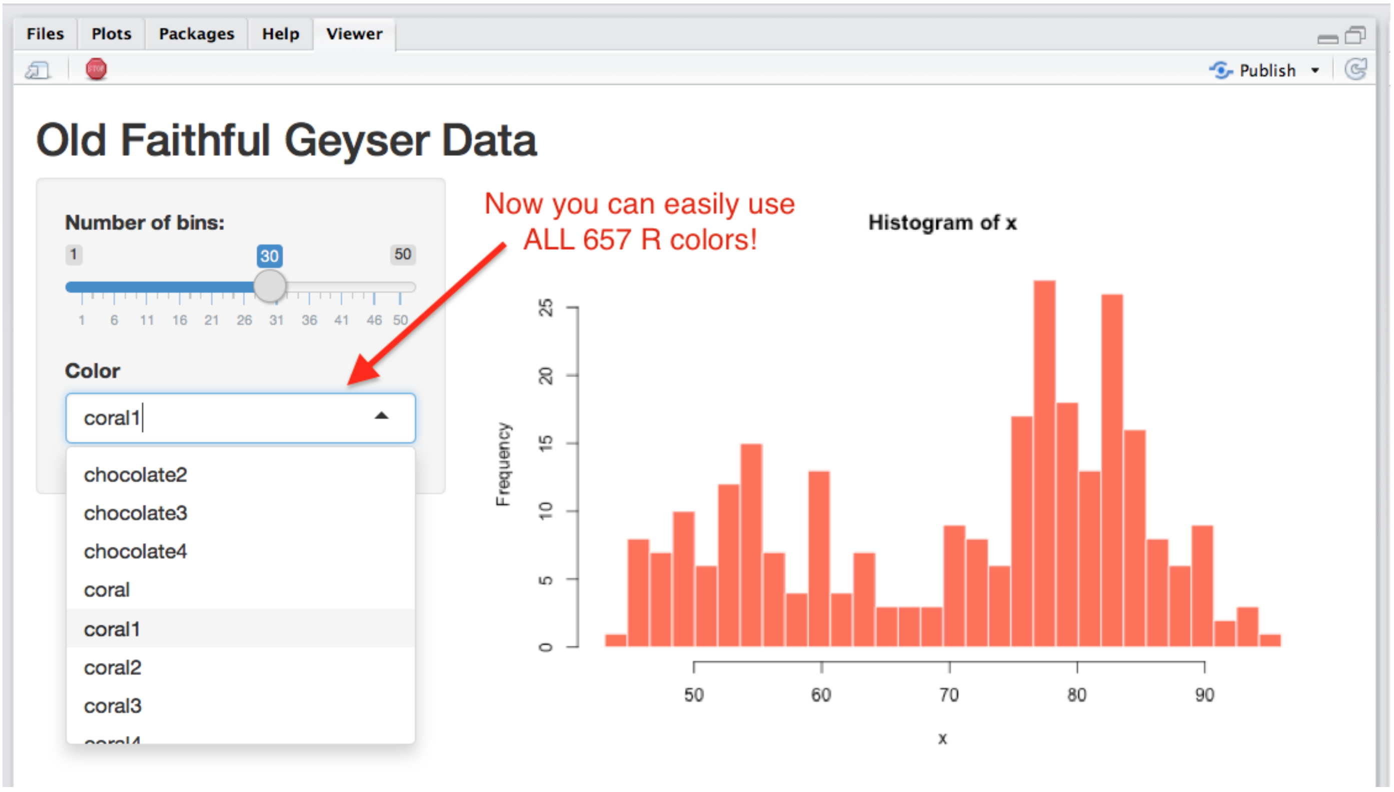
9. Right now we've told our color select input to have 4 possible colors by writing `choices = c("red", "blue", "green", and "black")`. But R has so many more colors, what if we wanted the select input to show all of the named colors in R like `"gold"`, `"skyblue"`, and more obscure colors like `"peachpuff4"`, `"olivedrab"` and `"lavender"`? It would be quite a pain to have to type all possible color names. Thankfully, we can easily pass a vector of color names to the `choices` input. The function `colors()` will return a vector of all 657 color names. In your console, try running `colors()` to see all of the colors.

```
# Print a vector of all color names
colors()
```

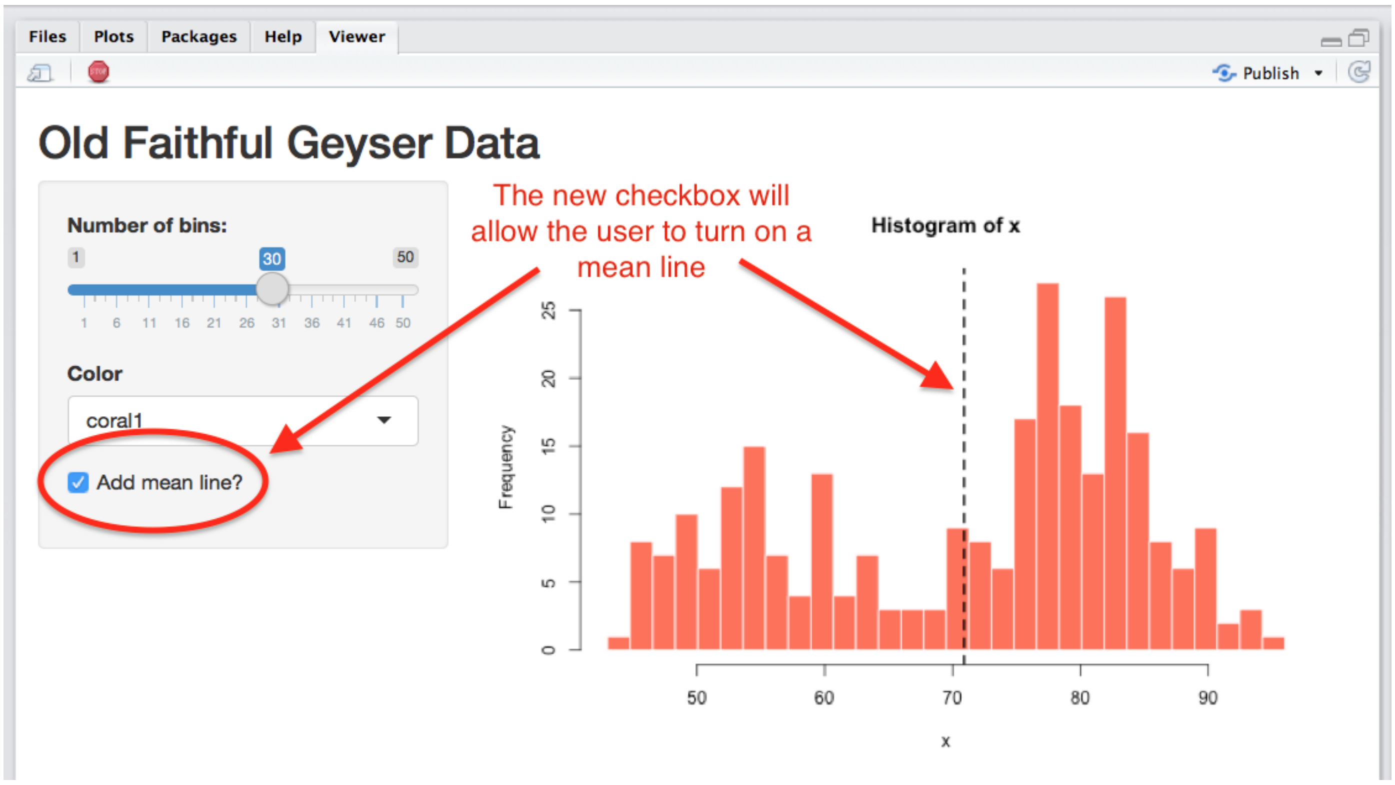
10. Now, all we need to do to allow the select input in our app to allow all of these colors is to include the argument `choices = colors()` as follows:

```
# This code will tell Shiny to include ALL 657 colors
#  in R as possible choices for the user in the bincolor input

 selectInput(inputId = "bincolor",
             label = "Color",
             choices = colors(), # <- HERE!!! change the choices to colors()
             selected = "black")
```

11. Now re-run your app! Your color input should now have access to *all* of the named colors in R. Try playing around with it to see the results!



Part C: Add a `checkboxInput` to add mean line



12. In this section, we'll add a new checkbox input that allows the user to add a line indicating the mean of the distribution. In the sidebar panel, add a new checkbox input called `"addmean"`

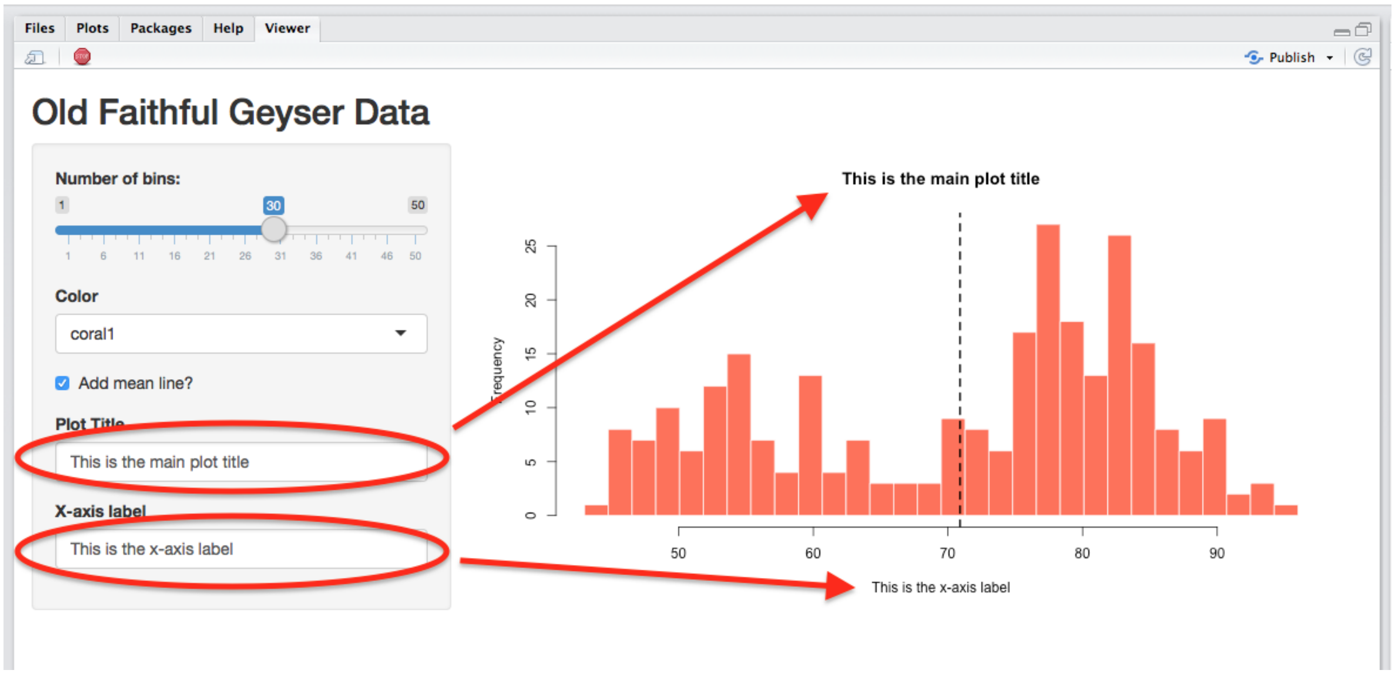
```
# Add a new checkboxInput to your app

checkboxInput(inputId = "addmean",
                  label = "Add mean line?",
                  value = FALSE)
```

13. Now it's time to include the actual plotting code that will show the line. Below the `hist()` code that creates the histogram, include the following:

```
# Add this code just below the histogram code to add
#   a line at the mean IF the addmean checkbox is checked.

if(input$addmean) {  # If the addmean box is checked...

    # Add a vertical line at the mean of x
    abline(v = mean(x),
           lwd = 2,       # Thickness
           lty = 2)       # Dashed line

  } # close if statement
```

14. Reload the app and see your new checkbox in action!

Part D: Add plot labels with `textInput`



Old Faithful Geyser Data

Number of bins: 30

Color: coral1

Add mean line? ✓

Plot Title: This is the main plot title

X-axis label: This is the x-axis label

15. Now we'll add the option for users to specify the title, and x-axis label of the plot. To do this, create two new text inputs using the `textInput()` function in the user interface.

```
# Add 2 text inputs to the sidepanel with textinput()

textInput(inputId = "main",
          label = "Plot Title"),

textInput(inputId = "xlab",
          label = "X-axis label")
```

16. Then, include these inputs in the `hist()` function in the server code:

```
hist(x,
     breaks = bins,
     col = input$bincolor,
     border = 'white',
     main = input$main,    # HERE!!! Change main title based on input$main
     xlab = input$xlab)    # HERE!!! Change x label  based on input$xlab
```

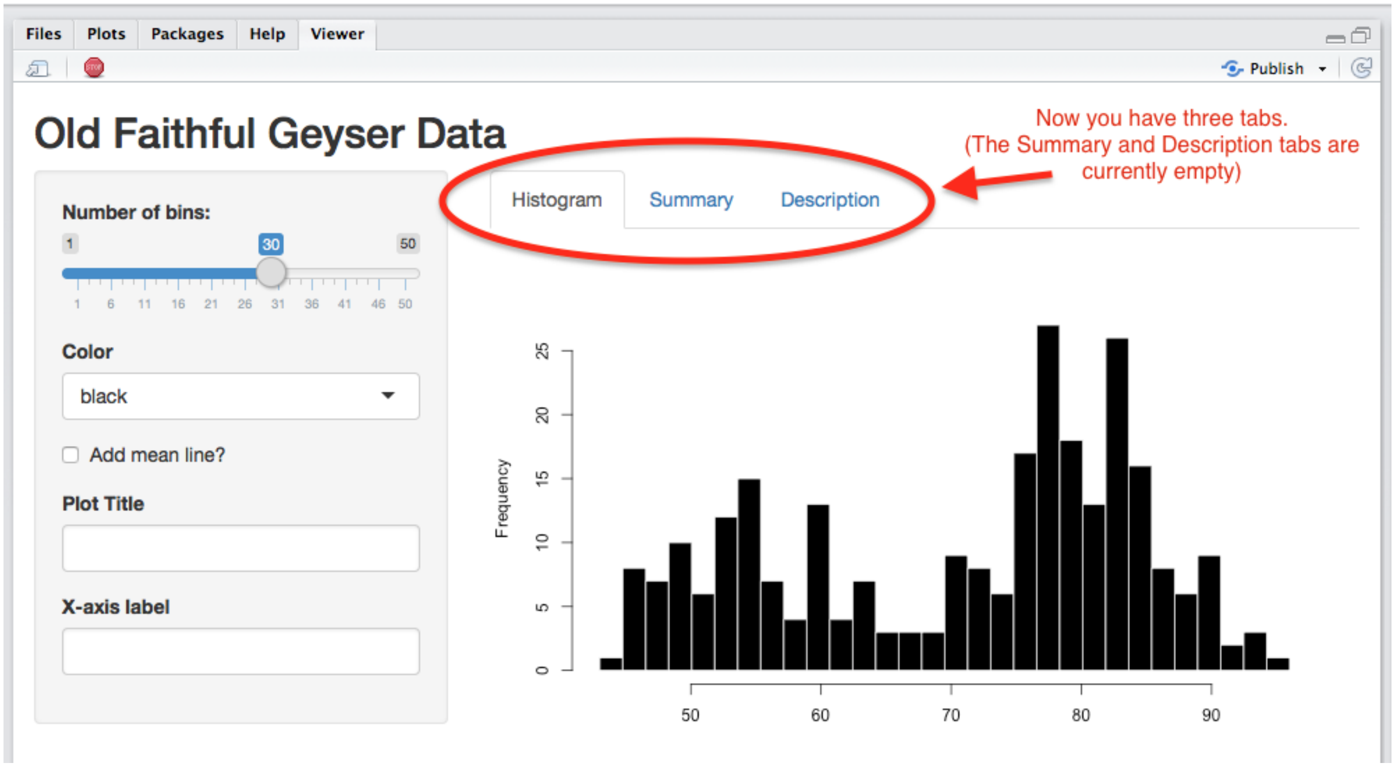
17. Reload the app and test your new text inputs!

Part E: Add a tab panel with `tabsetPanel`

18. One great way to create an app is to separate content, like different plots or tables of information, in different tabs. To create different tabs in your Shiny App, include the function `tabsetPanel()` within the `mainPanel()` function like this:

```
# Create different tabs within your app by using tabsetPanel()
#  Here, we'll create three tabs, where the first tab

mainPanel(
    tabsetPanel(type = "tabs",
                tabPanel("Histogram", plotOutput("distPlot")),  # The original Histogram
                tabPanel("Summary"),                            # Empty for now
                tabPanel("Description")                         # Empty for now
    )
    )
```

19. Reload your app, it should look like this. You should be able to click on the two new tabs, however, the Summary and Description tabs are currently empty. We'll fill them in the next step



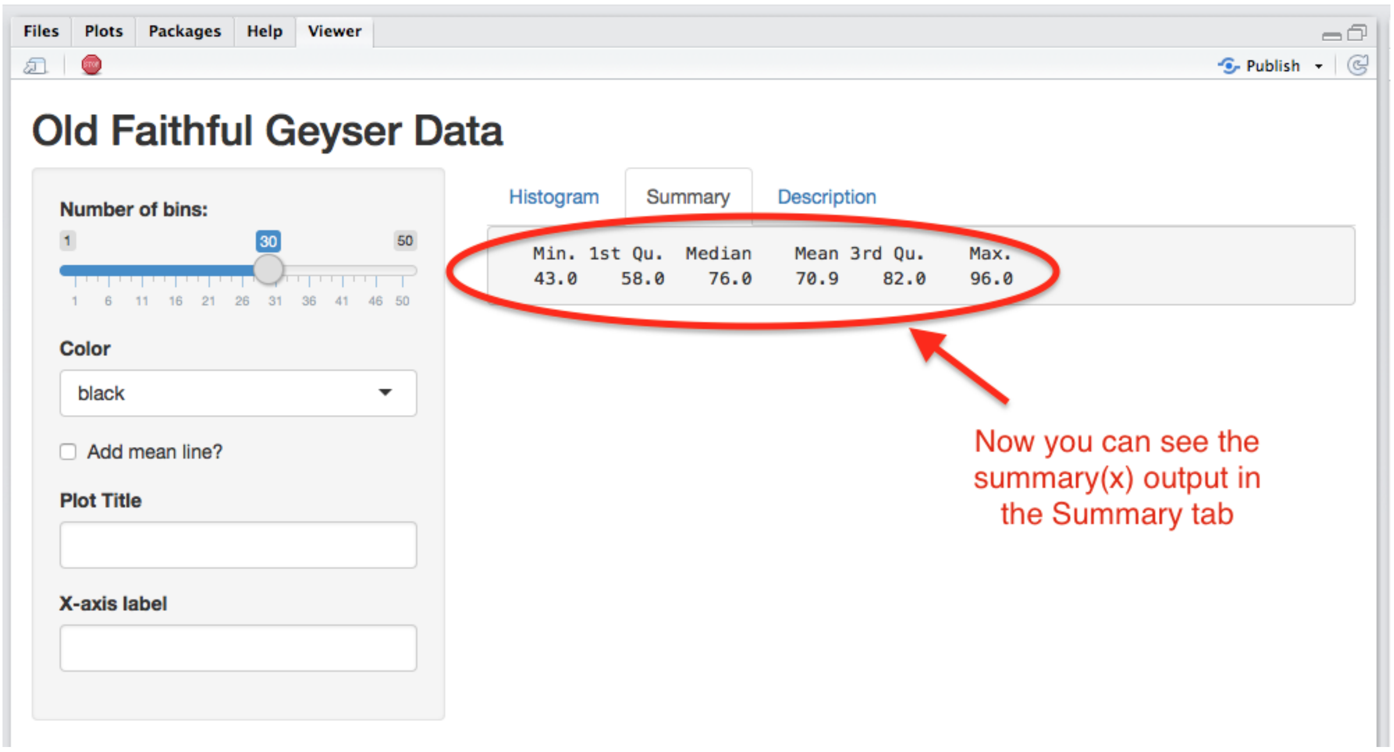
20. Now let's fill in the Summary tab. When the user clicks on this tab, we'll print summary statistics of the data. We'll do this in two steps. First, we need to create the summary statistics, then *render* those statistics using `renderPrint()`, and then assign the result to `output$summary`. To do this, add the following code within your server function (you can actually put it anywhere in the server function!)

```
# Add this code to the server function to return summary statistics

 output$summary <- renderPrint({
   x <- faithful[, 2]              # Define x again
   summary(x)
 })
```

21. Finally, we need to tell the `"Summary"` tab panel to display this information. To do this, add the function `verbatimTextOutput("summary")` to the `"Summary"` tab panel as follows:

```
# Add verbatimTextOutput("summary") to the Summary tab panel

tabsetPanel(type = "tabs",
            tabPanel("Histogram", plotOutput("distPlot")),
            tabPanel("Summary", verbatimTextOutput("summary")),   # <- HERE!!!
            tabPanel("Description")
)
```

22. Reload the app! You should now see the summary output in the Summary tab!

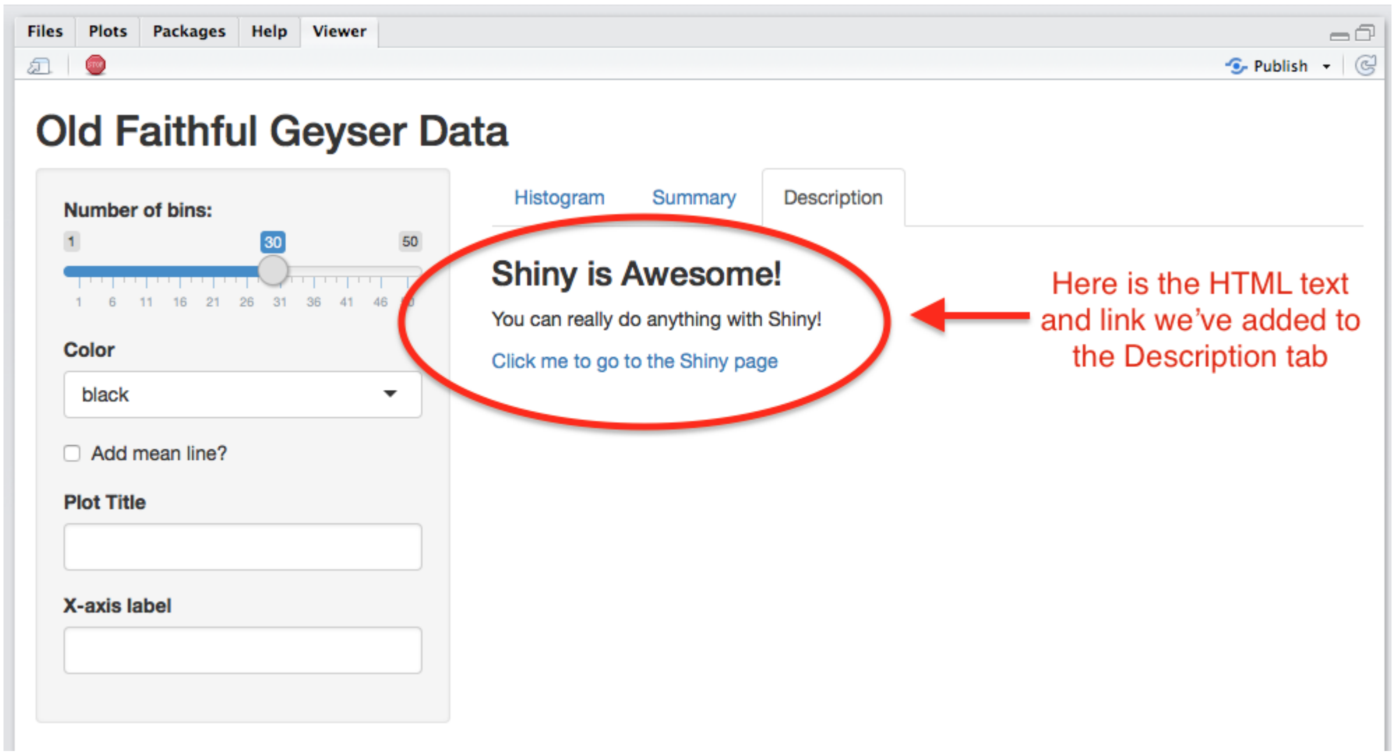


```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   43.0    58.0    76.0    70.9    82.0    96.0 
```

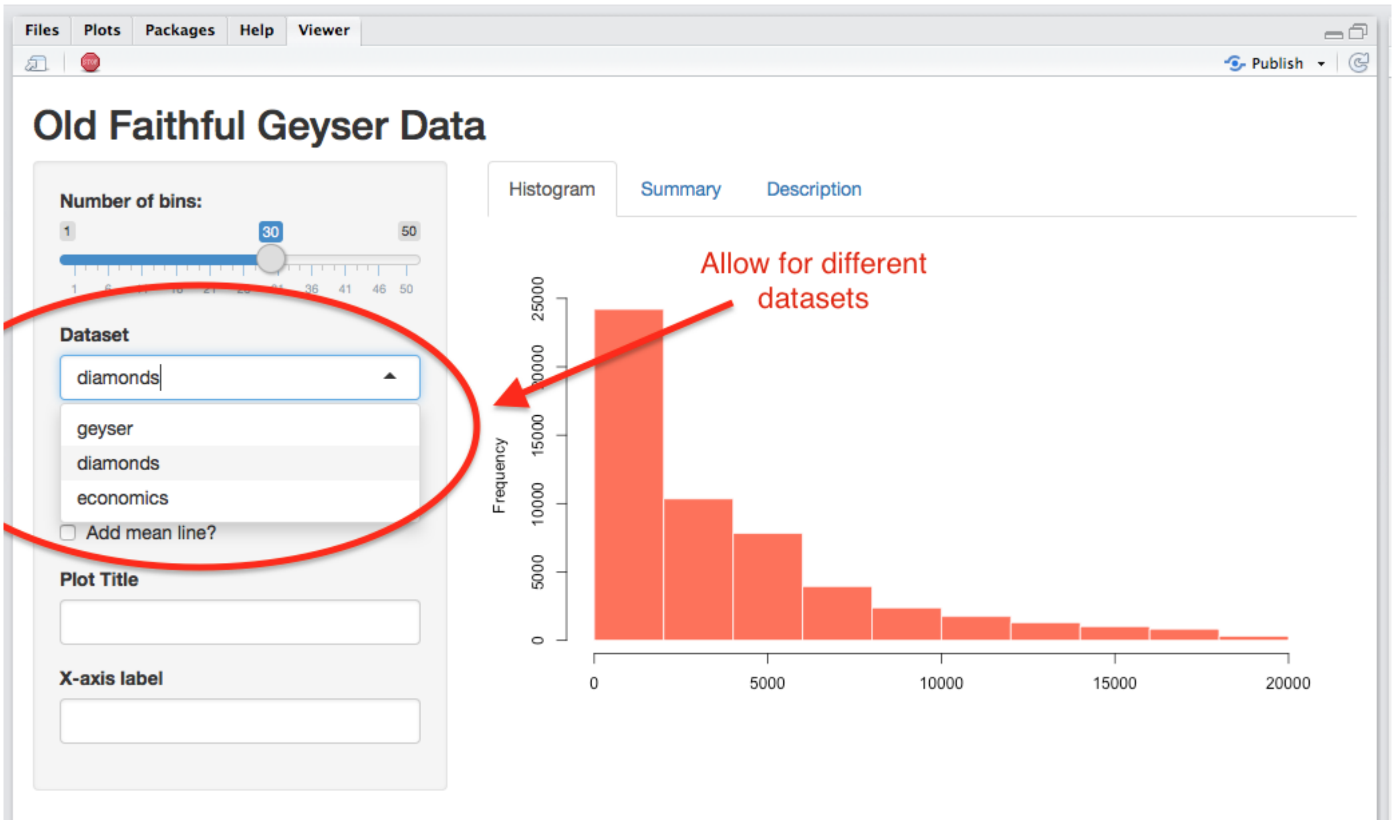
23. Finally, it's time to add a verbal description to our app in the Description tab. To do this, we can use a collection of the `h3(), p(), a()` functions. If you're familiar with HTML, these functions simply return HTML. Using the following code template, update the contents of the `"Description"` tab panel to display some text and a link.

```
# Add HTML functions to the Description tab

tabsetPanel(type = "tabs",
            tabPanel("Histogram", plotOutput("distPlot")),
            tabPanel("Summary", verbatimTextOutput("summary")),
            tabPanel("Description",
                     h3("Shiny is Awesome!"),
                     p("You can really do anything with Shiny!"),
                     a("Click me to go to the Shiny page",
                       href = "https://shiny.rstudio.com/gallery/")
            )
)
```

24. Reload your app and click on the Description tab, you should now see your text!



Part F: Access different datasets with `reactive()`



25. Next, we'll allow the user to plot data from 3 different datasets. To do this, first create a new select input box in the user interface as follows:

```
# Add a new select input called mydata

selectInput(inputId = "mydata",
            label = "Dataset",
            choices = c("geyser", "diamonds", "economics"),
            selected = "geyser")
```

26. Now, in your server section, we need to define `x` as a *reactive value* that depends on the user's input to `input$mydata`. To do this, we'll remove the existing code that defines the object `x` and replace it with

`x_reactive()` using the `reactive()` function. The `reactive()` function tells Shiny that `x_reactive` is isn't a stable value, but will change depending on user input.

- Step 1: Delete all existing code that defines the object `x` (right now you'll find it in both the `hist()` function and when we defined `output$summary`
- Step 2: Using the following code, define `x_reactive` as a reactive value that can change depending on the `input$mydata`. If you haven't seen it before, the `switch()` function is a very helpful function that works like a series of if-then statements.

```
# x is a reactive value that depends on input$mydata
#   This code should replace the previous code that defined x

# Put this code just after the server() function and BEFORE output$distPlot

x_reactive <- reactive({

  switch(input$mydata,  # Look at the value of input$mydata...
         "geyser" = faithful[,2],          # If the value is 'geyser', return data from faithful
         "economics" = economics$unemploy, # if the value is 'economics' return unemployment data
         "diamonds" = diamonds$price)      # If the value is 'diamonds' return price data

})
```

27. Now we need to make one last change. In **all** of the code that refers to `x` we need to replace `x` with `x_reactive()` (Don't forget the parentheses!!). Make this replacement for `hist()`, `summary()`, and `abline()`. Here is how you can do it:

```
# Change the value x to x_reactive() in seq(), hist(), abline() and summary()

bins <- seq(min(x_reactive()),   # HERE!!! Change to x_reactive()
            max(x_reactive()),   # HERE!!! Change to x_reactive()
            length.out = input$bins + 1)

# [Skip some code...]

 hist(x_reactive(),              # HERE!!! Change to x_reactive()
      breaks = 10,
      col = input$bincolor,
      border = 'white',
      main = input$main,
      xlab = input$xlab)

# [Skip some code...]

 abline(v = mean(x_reactive()),   # HERE!!! Change to x_reactive()
        lwd = 2,
        lty = 2)

# [Skip some code...]

output$summary <- renderPrint({
   summary(x_reactive())         # HERE!!! Change to x_reactive()
 })
```

28. Reload your app and have fun!! You're finished!

Challenges

C1. Now that you have multiple datasets, try removing the Plot title and X-axis label inputs (`input$main`, `input$xlab`), and instead have the plot automatically adjust the plot title and axis label based on the dataset the users selects. Just like we did with `x_reactive()`, you'll need to assign the main title and x label as reactive values.

C2. Instead of a histogram, try creating a scatterplot as the main plotting output. Keep in mind that you'll now need to create two reactive values: `x_reactive()` for the data on the x-axis and `y_reactive()` for the y-axis. Then, as an optional argument (using a `checkboxInput`), allow the user to add a regression line to the plot.

C3. Check out the RStudio Shiny Gallery (https://shiny.rstudio.com/gallery/) at https://shiny.rstudio.com/gallery/ (https://shiny.rstudio.com/gallery/) and look for a Shiny app you really like. Find the code, copy it, and paste it into a new app in RStudio. See if you can get the app to run. Then, customise it!

References

- For more examples of apps in action, check out the RStudio Shiny Gallery (https://shiny.rstudio.com/gallery/) at https://shiny.rstudio.com/gallery/ (https://shiny.rstudio.com/gallery/)