

Building Shiny Apps



Dean Attali
Shiny consultant

@daattali
attalitech.com

Open Data Science Conference
San Francisco
Oct 29, 2019

Supplemental exercise files available at
<https://github.com/daattali/shiny-workshop-odsc2019>

- Interactive hands-on workshop, NOT a lecture
- Ask questions
- If something is unclear - ask me to explain!
- Resources: your neighbour > Google > me

A gray slide means you have work to do!



Your turn

Introduce yourself to your neighbours
on both sides



Your turn

Shiny is an R package that makes it easy to **build web applications** with R



“Building web application” sounds scary



Not with shiny - 0 knowledge of web technologies required!

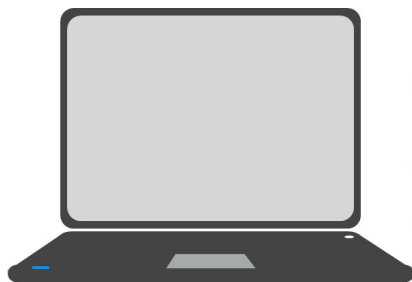
- Interactively explore data <https://daattali.com/shiny/user2017/>
- Just for fun <https://daattali.com/shiny/lightsout/>
- Easy interface to run an R analysis <https://daattali.com/shiny/ddpcr/>
- Complete websites <https://cranalerts.com>
- Many examples by users <http://ShowMeShiny.com>
- What we'll build: Visualizing NBA 2018/19 Player Stats
<https://daattali.shinyapps.io/nba2018/> OR
<https://daattali.com/shiny/nba2018/>

Explore examples, get a feel for what shiny can do

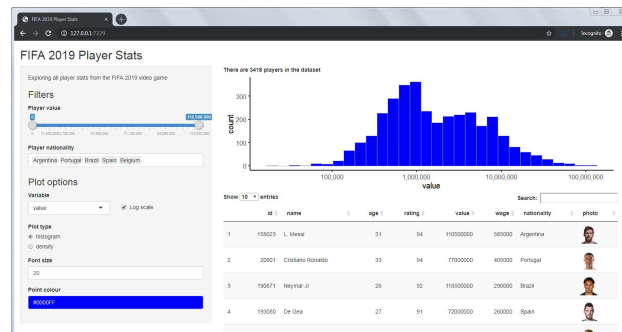


Your turn

What is a Shiny app?

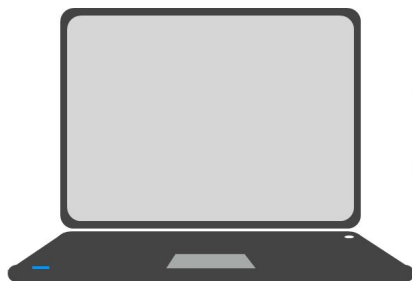


Computer
running R

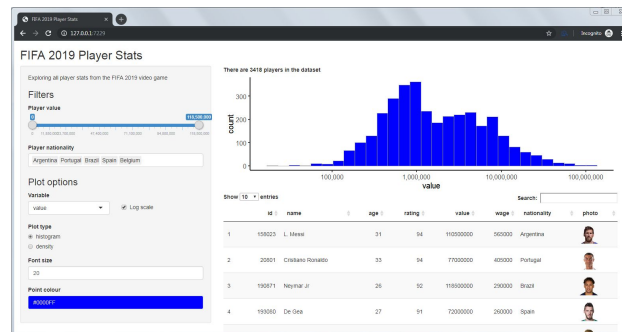


Web page

What is a Shiny app?



Server code



User interface (UI)

Shiny app template

```
library(shiny)

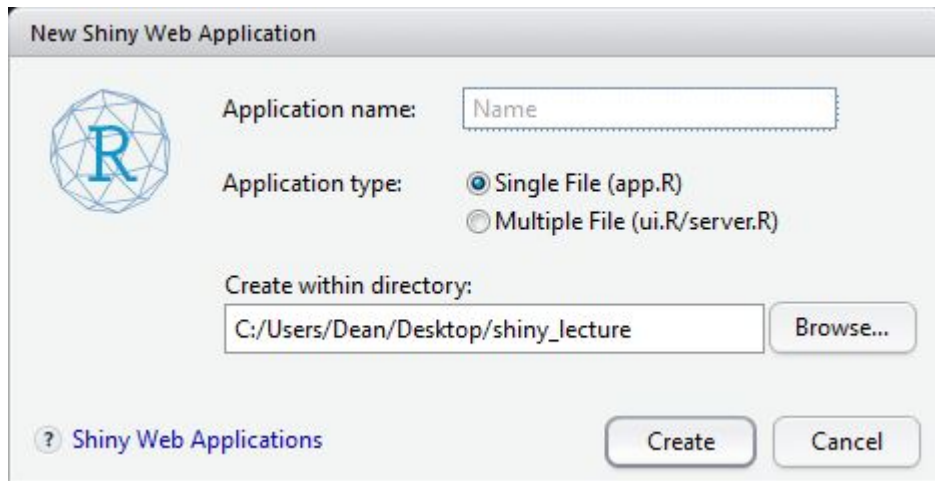
ui <- fluidPage()

server <- function(input, output) {}

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

A little cheat..

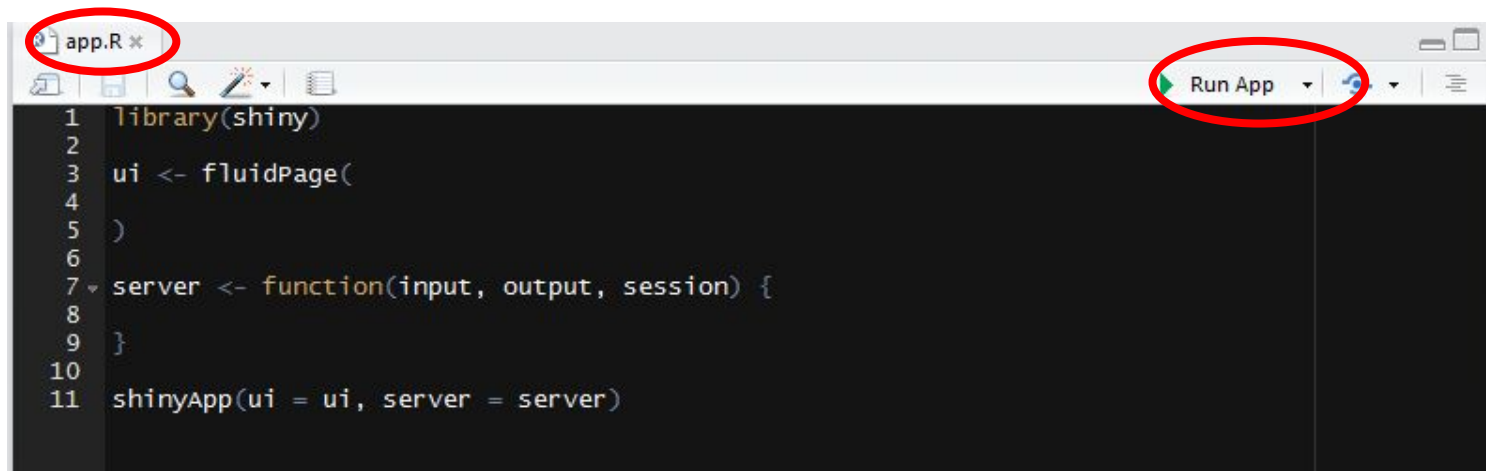
File > New File > Shiny Web App...



Or use RStudio Snippets: type “shiny” and select “shinyapp” from the autocomplete menu

Run Shiny app in RStudio

Save file as “**app.R**” → “*Run App*”



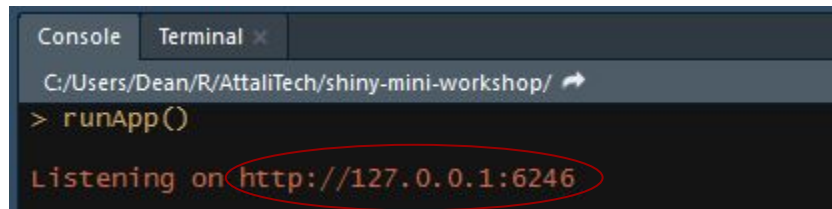
The screenshot shows the RStudio interface. The top-left pane displays a file named `app.R`, which is circled in red. The top-right pane shows the `Run App` button, also circled in red. The main editor pane contains the following R code:

```
1 library(shiny)
2
3 ui <- fluidPage(
4   )
5
6
7 server <- function(input, output, session) {
8
9 }
10
11 shinyApp(ui = ui, server = server)
```



Do not place any code after `shinyApp(...)`

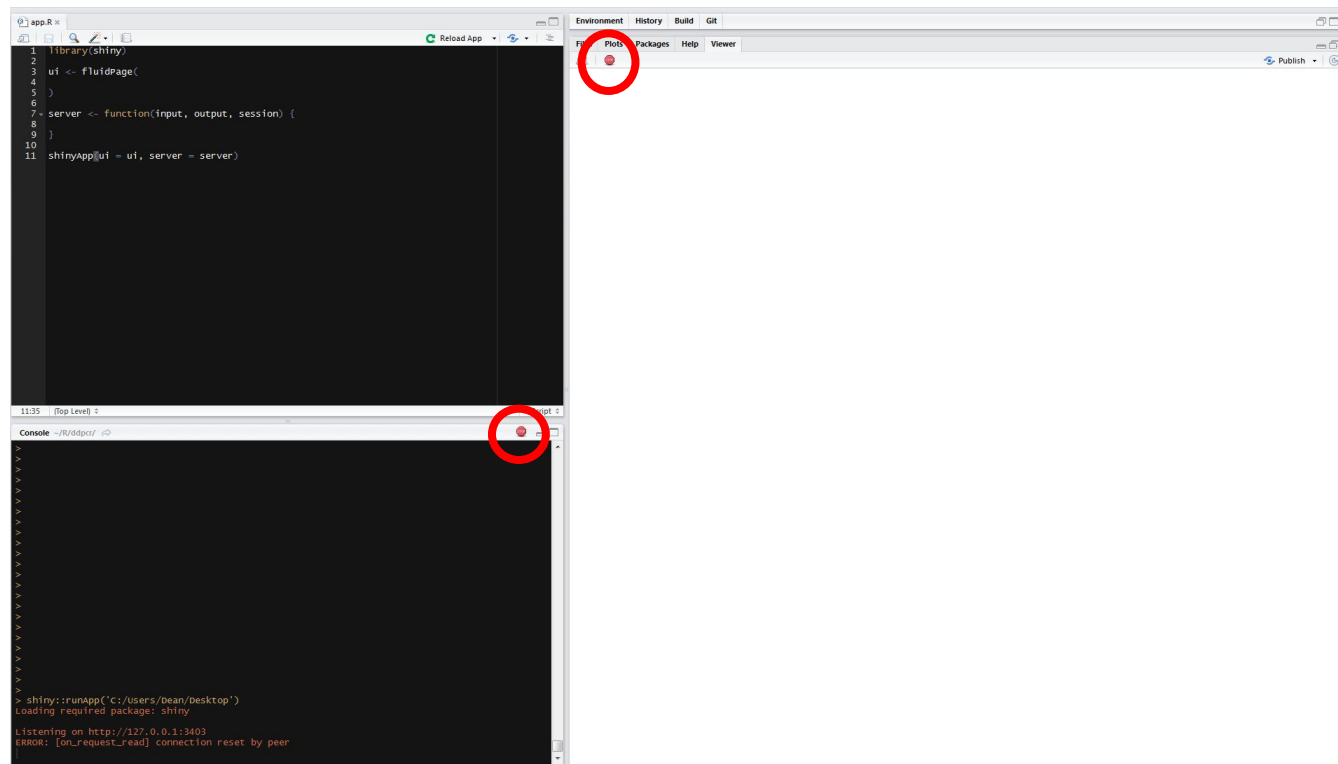
R session now busy - running the Shiny app



```
Console Terminal x  
C:/Users/Dean/R/AttaliTech/shiny-mini-workshop/ ↗  
> runApp()  
Listening on http://127.0.0.1:6246
```

Console tells you URL for app - can open in browser

Stop Shiny app in RStudio



Press
Escape
or click
the *Stop*
icon

- Load players dataset after loading shiny:

```
players <- read.csv("data/nba2018.csv")
```

- Verify data loaded: show the number of rows in the UI of the app
- Explore the data for a couple minutes. What are the age ranges of players? How about salary? Heights?



Your turn

app_01.R



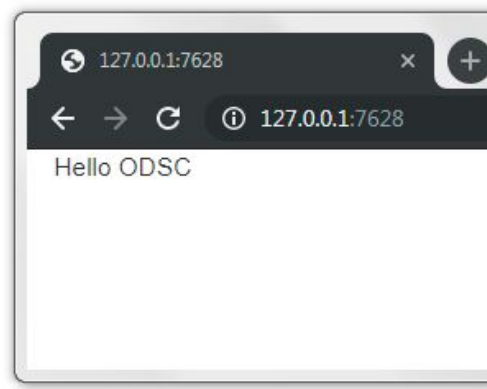
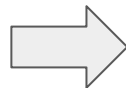
Add text as argument to `fluidPage()`

```
library(shiny)

ui <- fluidPage(
  "Hello ODSC"
)

server <- function(input, output) {}

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



`fluidPage()` accepts arbitrary number of arguments

Formatted text

h1 () **Primary header**

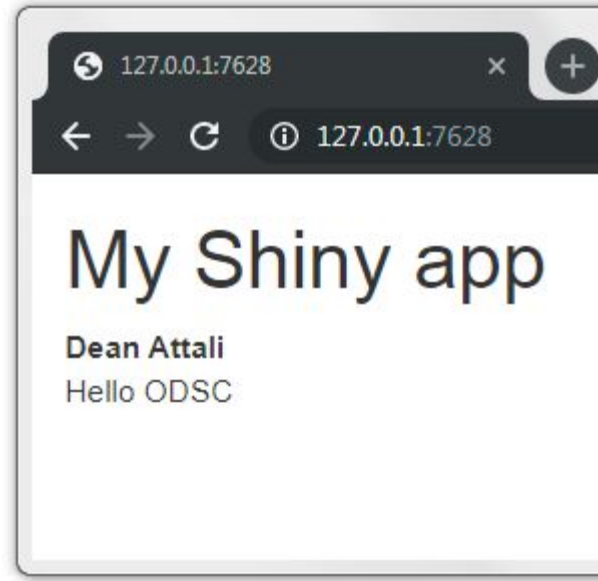
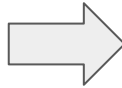
h2 () Secondary header

strong () **Bold**

em () *Italicized (emphasized)*

br () Line break

```
ui <- fluidPage(  
  h1("My Shiny app"),  
  strong("Dean Attali"),  
  br(),  
  "Hello ODSC"  
)
```



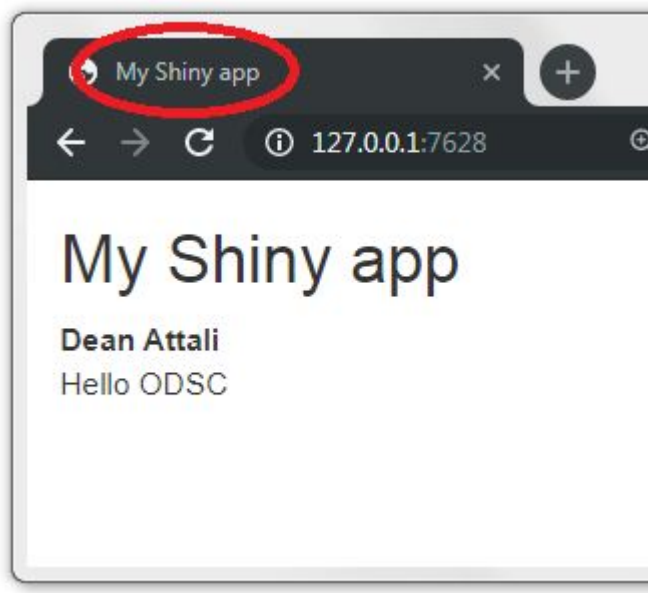
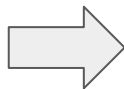
Watch out for commas!

If you know HTML, `tags` is a list with all HTML tags

```
> names(tags)
[1] "a"          "abbr"      "address"   "area"      "article"
[6] "aside"     "audio"     "b"         "base"      "bdi"
[11] "bdo"       "blockquote" "body"      "br"        "button"
[16] "canvas"    "caption"   "cite"      "code"      "col"
[21] "colgroup"  "command"   "data"      "datalist"  "dd"
[26] "del"       "details"   "dfn"       "div"       "dl"
[31] "dt"        "em"        "embed"     "eventsource" "fieldset"
[36] "figcaption" "figure"    "footer"    "form"      "h1"
[41] "h2"        "h3"        "h4"        "h5"        "h6"
[46] "head"      "header"    "hgroup"    "hr"        "html"
[51] "i"         "iframe"    "img"       "input"     "ins"
[56] "kbd"       "keygen"    "label"     "legend"    "li"
[61] "link"      "mark"      "map"       "menu"      "meta"
[66] "meter"     "nav"       "noscript"  "object"    "ol"
[71] "optgroup"  "option"    "output"    "p"         "param"
[76] "pre"       "progress"  "q"         "ruby"      "rp"
[81] "rt"        "s"         "samp"      "script"    "section"
[86] "select"    "small"     "source"    "span"      "strong"
[91] "style"     "sub"       "summary"   "sup"       "table"
[96] "tbody"     "td"        "textarea"  "tfoot"     "th"
[101] "thead"    "time"      "title"     "tr"        "track"
[106] "u"        "ul"        "var"       "video"     "wbr"
```

For title, use `titlePanel()` instead of `h1()`

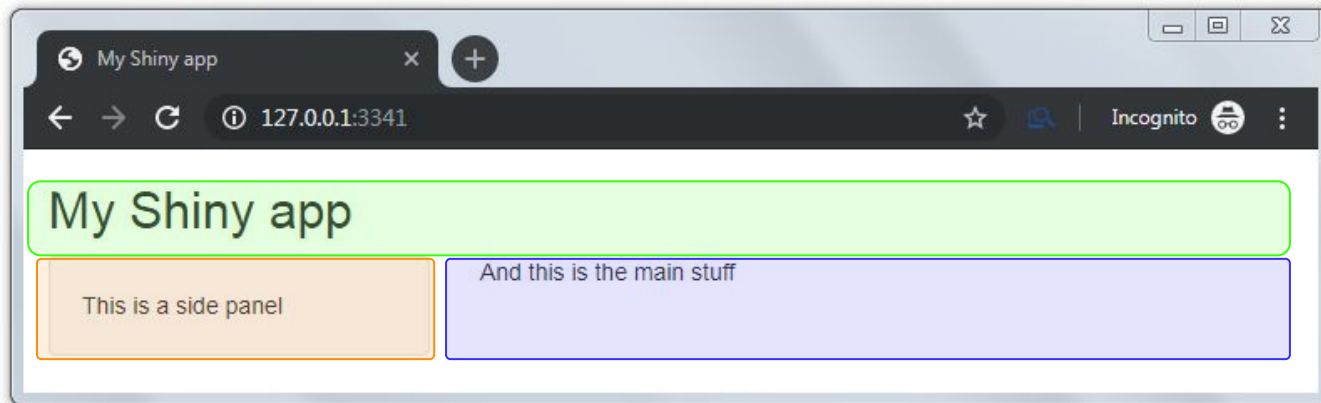
```
ui <- fluidPage(  
  titlePanel("My Shiny app"),  
  strong("Dean Attali"),  
  br(),  
  "Hello ODSC"  
)
```



Layouts

- By default, all elements stack up one after the other
- Layout adds control over positioning
- Layout options: menu bars, tabs, rows + columns, ...
<http://shiny.rstudio.com/articles/layout-guide.html>
- Most popular beginner layout: `sidebarLayout()`

```
fluidPage(  
  titlePanel("My Shiny app"),  
  sidebarLayout(  
    sidebarPanel(  
      "This is a side panel"  
    ),  
    mainPanel(  
      "And this is the main stuff"  
    )  
  )  
)
```

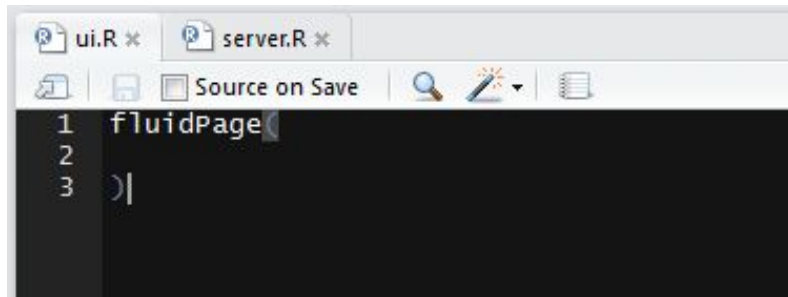


UI functions are simply HTML wrappers

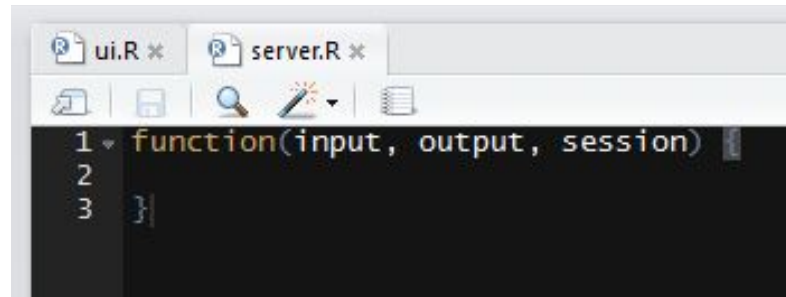
```
> print(ui)
<div class="container-fluid">
  <h2>My Shiny app</h2>
  <div class="row">
    <div class="col-sm-4">
      <form class="well">This is a side panel</form>
    </div>
    <div class="col-sm-8">And this is the main stuff</div>
  </div>
</div>
```

Run Shiny app in RStudio - method 2

Save UI as “**ui.R**”, server as “**server.R**”




```
1 fluidPage{  
2  
3 }
```



```
1 function(input, output, session) {  
2  
3 }
```

Good for complex Shiny apps, separates view vs logic

 **Do not call** `shinyApp(...)`

- Add a title of “NBA 2018/19 Player Stats”
- Add a sidebar layout
 - Side panel should have the text “Exploring all player stats from the NBA 2018/19 season”
 - Main panel should have the text “There are X players in the dataset” **in bold** (use the real number for X)
- Bonus: Explore the different arguments of `titlePanel()`, `sidebarLayout()`, `sidebarPanel()`, `mainPanel()`



Your turn

app_02.R



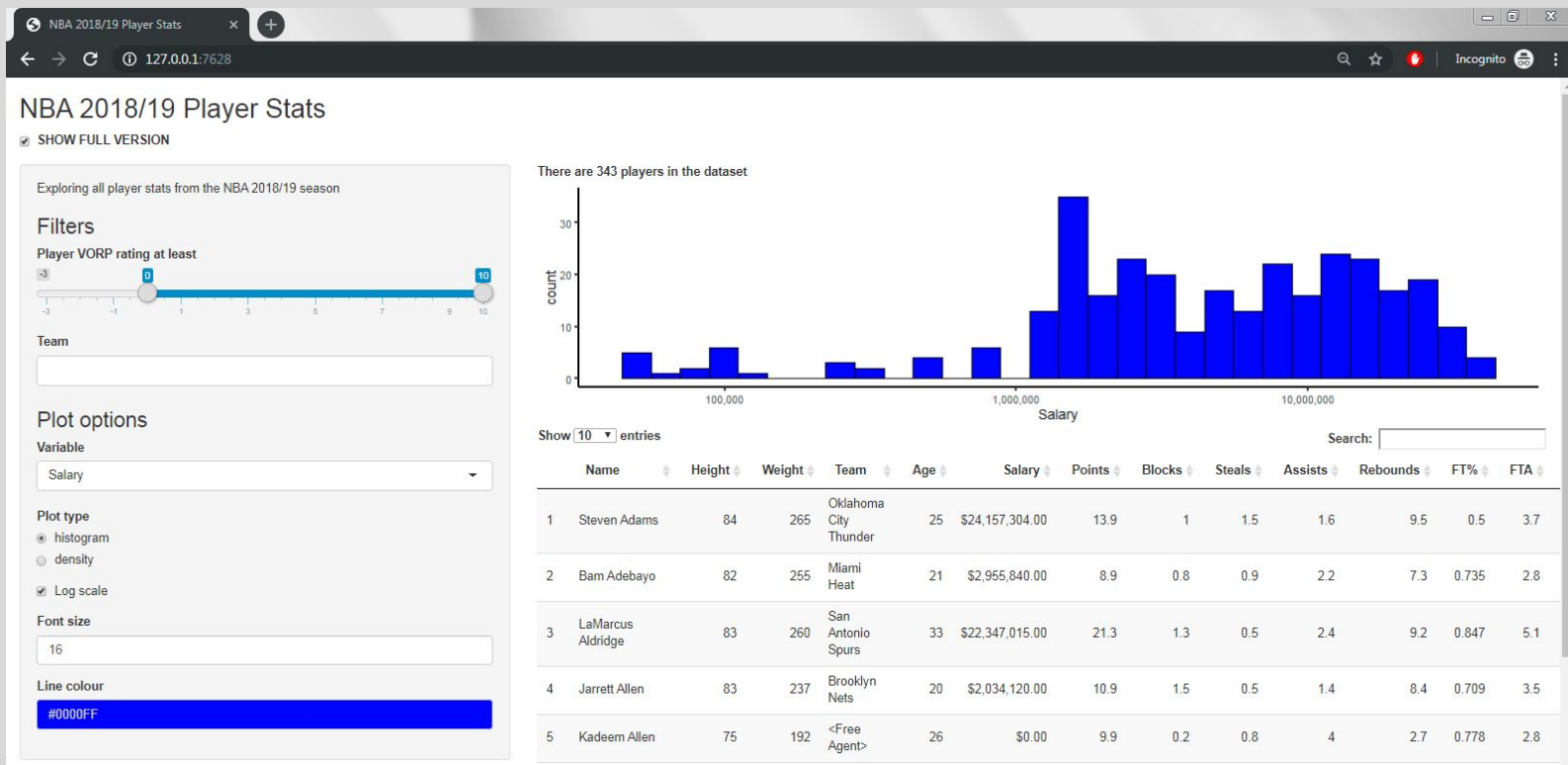
Inputs and outputs

- For interactivity, app needs inputs and outputs
- **Inputs** - things user can toggle
- **Output** - R objects user can see, often depend on inputs

```
fluidPage(  
  # *Input() functions,  
  # *Output() functions  
)
```

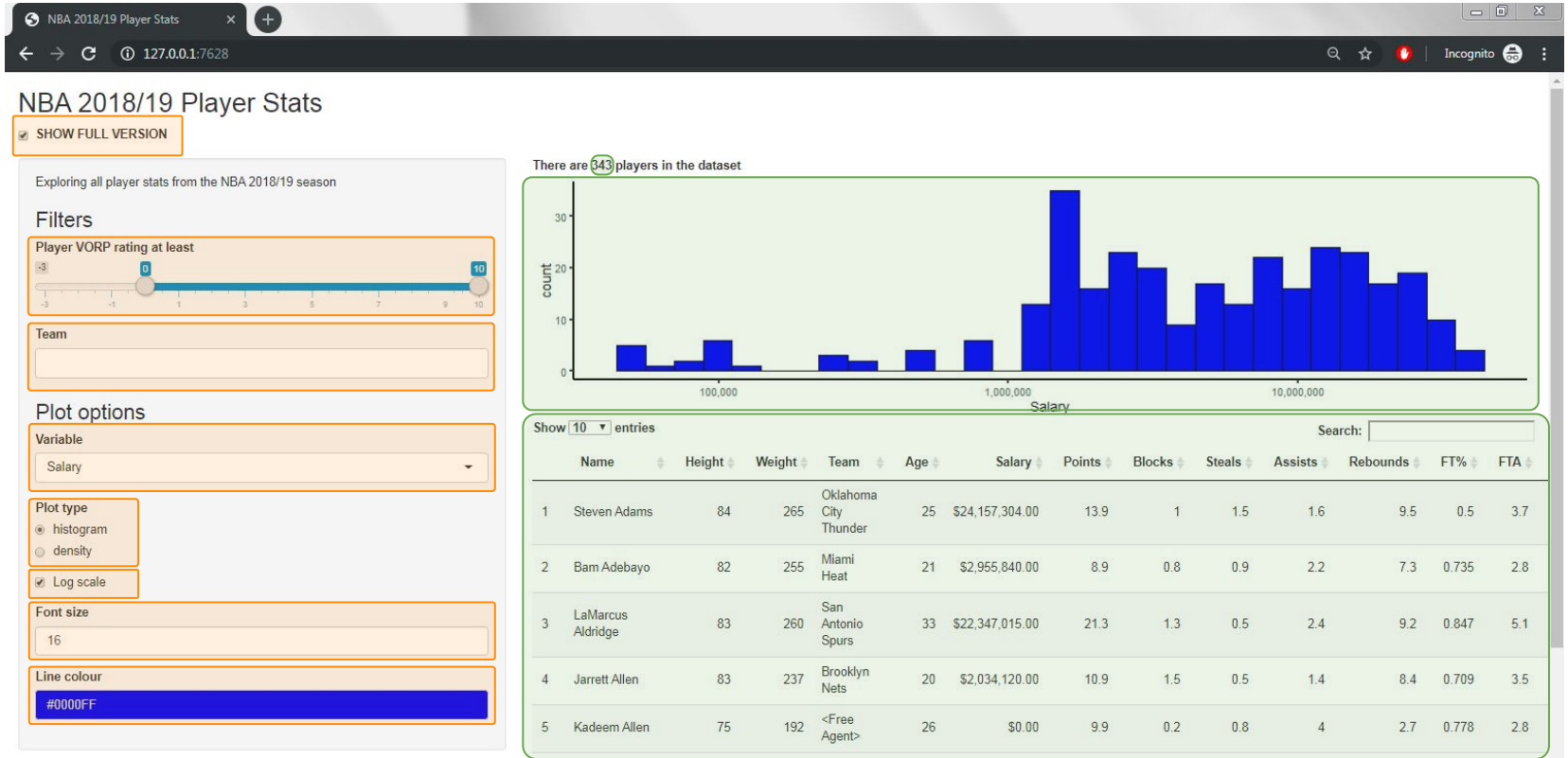
ui.R

How many inputs? How many outputs?



Your turn

8 inputs, 3 outputs



Buttons

Action

Submit

`actionButton()`
`submitButton()`

Date range

2014-01-24 to 2014-01-24

`dateRangeInput()`

Radio buttons

- ☒ Choice 1
☐ Choice 2
☐ Choice 3

`radioButtons()`

Single checkbox

☒ Choice A

`checkboxInput()`

File input

Choose File No file chosen

`fileInput()`

Select box

Choice 1

`selectInput()`

Checkbox group

- ☒ Choice 1
☐ Choice 2
☐ Choice 3

`checkboxGroupInput()`

Numeric input

1

`numericInput()`

Sliders



`sliderInput()`

Date input

2014-01-01

`dateInput()`

Password Input

.....

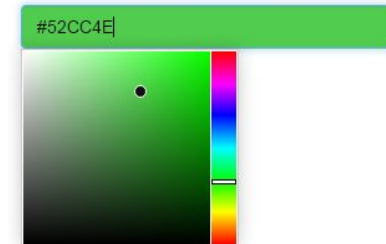
`passwordInput()`

Text input

Enter text...

`textInput()`

Colour input



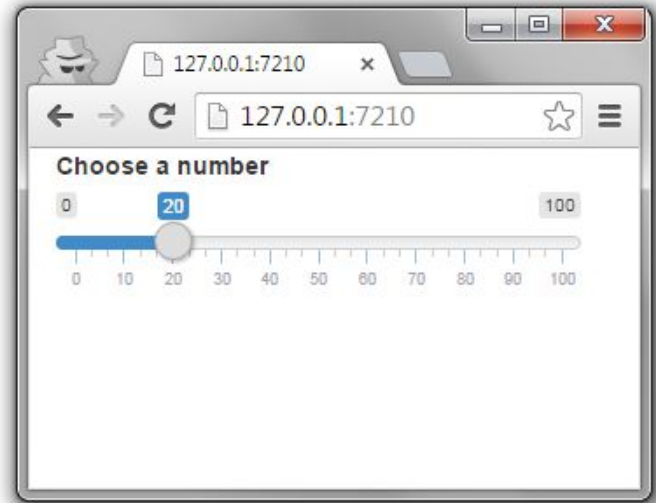
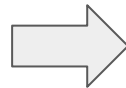
`colourpicker::colourInput()`

```
library(shiny)

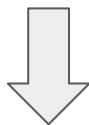
ui <- fluidPage(
  sliderInput(
    inputId = "num",
    label = "Choose a number",
    min = 0, max = 100,
    value = 20
  )
)

server <- function(input, output) {}

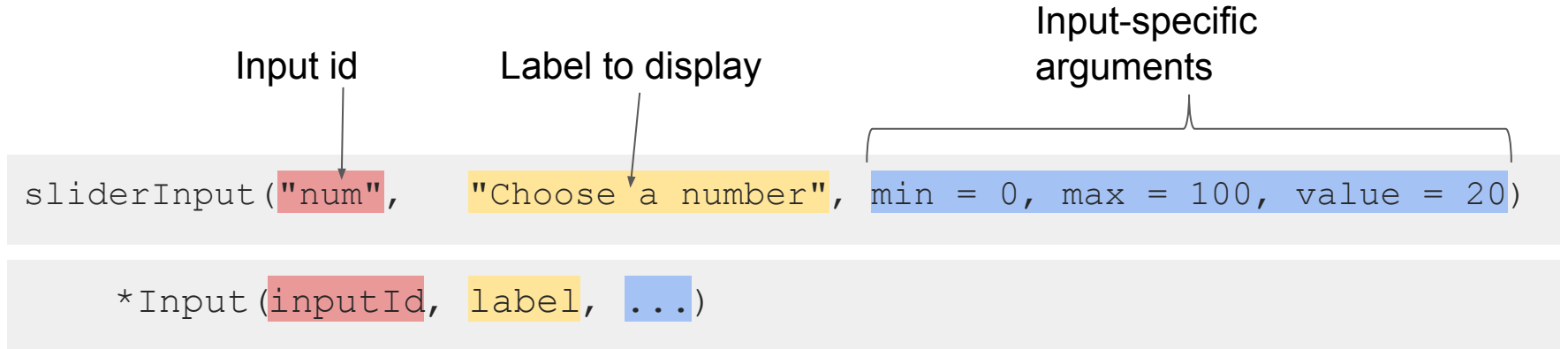
shinyApp(ui = ui, server = server)
```




```
sliderInput("num", "Choose a number",  
            min = 0, max = 100, value = 20)
```



```
<div class="form-group shiny-input-container">  
  <label class="control-label" for="num">Choose a number</label>  
  <input class="js-range-slider" id="num" data-min="0" data-max="100"  
data-from="20" data-step="1" data-grid="true" data-grid-num="10"  
data-grid-snap="false" data-prettify-separator="," data-prettify-enabled="true"  
data-keyboard="true" data-data-type="number"/>  
</div>
```



What arguments can I pass to an input function?

```
?sliderInput
```

- Add a level 3 header (using `h3()`) to the sidebar with the text “Filters”
- Add a slider input with an ID of “VORP”, possible values ranging from -3 to 10, default value of 0, and a label of “Player VORP rating at least”
- Add a dropdown selector (using `selectInput()`) with an ID of “Team”, “Golden State Warriors” as the default selection, and all teams in the dataset as possible choices.



Your turn

app_03.R



Outputs: Plots, tables, text - anything that R creates and users see

Two steps:

1. Create placeholder for output, in UI
2. Write R code to generate output, in server

UI Function	Outputs
<code>plotOutput()</code>	plot
<code>tableOutput()</code>	table
<code>textOutput()</code>	text
<code>uiOutput()</code>	Shiny UI element

Output name Output-specific arguments

```
plotOutput("myplot", width = "300px")
```

```
*Output(outputId, ...)
```

What arguments can I pass to an output function?

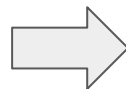
```
?plotOutput
```

```
library(shiny)

ui <- fluidPage(
  sliderInput("num", "Choose a number",
             0, 100, 20),
  plotOutput("myplot")
)

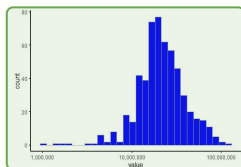
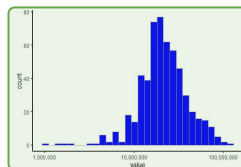
server <- function(input, output) {}

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



Summary

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



Begin app with template

Add elements as arguments to **fluidPage()**

Create inputs with ***Input()** functions

Create outputs with ***Output()** functions

Use **server** to assemble inputs into outputs

- Add a plot output placeholder in the main panel, with an ID of “nba_plot”
- Add a table output placeholder after the plot, with an ID of “players_data”
- Remember you won’t actually see anything change!



Your turn

app_04.R



Server is where outputs are built and sent to the UI

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

Input is a list to read values from (the inputs from the user)

Output is a list to write R objects (plots, tables, etc) into

Building outputs: 3 rules

```
server <- function(input, output) {  
  
  output$myplot <- renderPlot({  
    plot(rnorm(input$num))  
  })  
  
}
```

Building outputs

1 - Build object inside render function

```
server <- function(input, output) {  
  
  output$myplot <- renderPlot({  
    plot(rnorm(input$num))  
  })  
  
}
```

`*Output() → render*()`

Output function	Render function
<code>plotOutput()</code>	<code>renderPlot({})</code>
<code>tableOutput()</code>	<code>renderTable({})</code>
<code>textOutput()</code>	<code>renderText({})</code>
<code>uiOutput()</code>	<code>renderUI({})</code>

Building outputs

2 - Save object to `output$<id>`

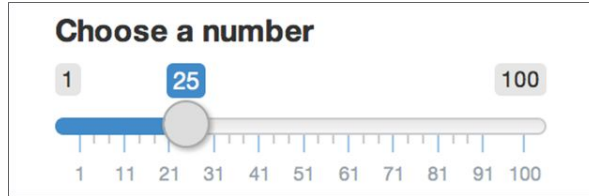
```
server <- function(input, output) {  
  
  output$myplot <- renderPlot({  
    plot(rnorm(input$num))  
  })  
  # in UI: plotOutput("myplot")  
}
```

Building outputs

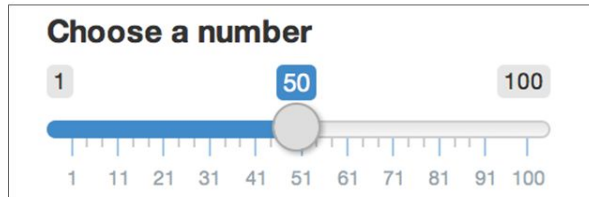
3 - Access input values with `input$id`

```
server <- function(input, output) {  
  output$myplot <- renderPlot({  
    plot(rnorm(input$num))  
  
    # in UI: sliderInput("num", ...) )  
  })  
}
```

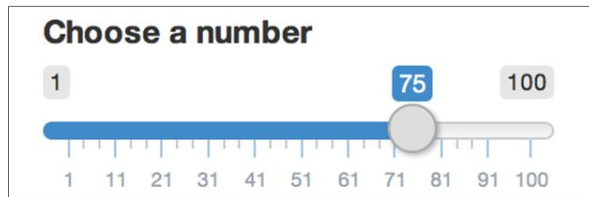
Using input\$



`input$num` returns 25



`input$num` returns 50



`input$num` returns 75

- Build the text output `num_players` that shows how many players are currently filtered. Use the same filtering code from the table.
- Build the plot output. It should be a histogram of player salaries, based on the filtered players data. You can use the same filtering code that the table output uses. The histogram can be constructed as

```
ggplot(data, aes(Salary)) + geom_histogram()
```
- Bonus: The `DT` package provides interactive tables. Change the table to use a DT table instead. Hint: only the UI placeholder function and the render function need to change



Your turn

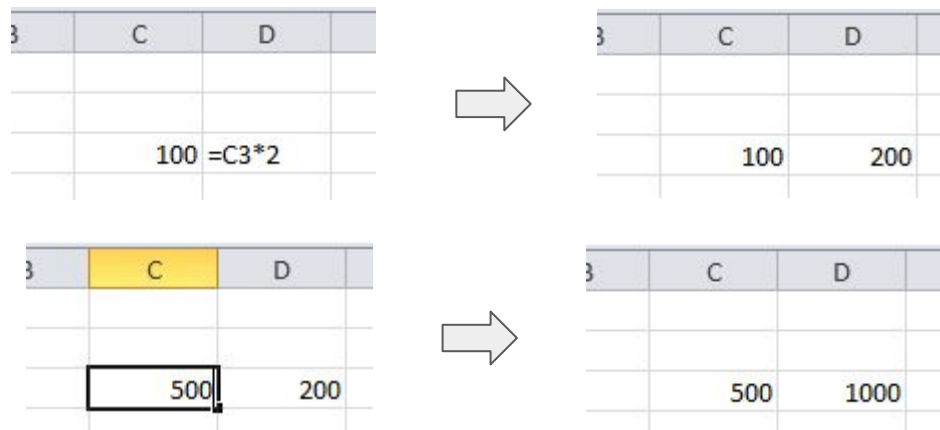
app_05.R



Reactivity

- Shiny uses **reactive programming**
- Supports **reactive variables**
- Allows outputs to automatically **react** to changes in inputs
- When value of variable x changes, anything that relies on x is re-evaluated

Spreadsheets are reactive



Regular R is not reactive

```
x <- 5
y <- x + 1
x <- 10
# What is y? 6 or 11?
```

- All inputs are **reactive**, so can always use in render functions

```
output$myplot <- renderPlot({  
  plot(rnorm(input$num) )  
})
```

- `output$myplot` **depends on** `input$num`
 - `input$num` **changes** → `output$myplot` **reacts**

```
fluidPage(  
  numericInput("x", "X", 5),  
  numericInput("y", "Y", 10),  
  textOutput("sum")  
)
```

ui.R

```
function(input, output) {  
  output$sum <- renderText({  
    input$x + input$y  
  })  
}
```

server.R

When does the output get re-rendered?

1. When the user changes the value of x , but not when y changes
2. When the user changes the value of y , but not when x changes
3. When the user changes the value of either x or y
4. Only after the user changes the values of both x and y



Your turn

- Reactive values can only be used inside **reactive contexts**
- Any `render*` function is a reactive context
- Accessing reactive value outside of reactive context: ERROR

```
server <- function(input, output) {  
  print(input$num)  
}  
# ERROR: Operation not allowed without an active reactive context.
```

- `observe ({ ... })` to **access** reactive variables

```
server <- function(input, output) {  
  observe({  
    print(input$num)  
  })  
}
```

- Each reactive variable creates a dependency

```
server <- function(input, output) {  
  observe({  
    print(input$num1)  
    print(input$num2)  
  })  
}
```

```
server <- function(input, output) {  
  x <- input$num + 1  
}  
# ERROR: Operation not allowed without an active reactive context.
```

`reactive({ ... })` to **create** reactive variables

```
server <- function(input, output) {  
  x <- reactive({  
    input$num + 1  
  })  
}
```



`reactive()` variables must be accessed with parentheses

- Duplicated code \Rightarrow multiple places to maintain
 - When code needs updating
 - When bugs need fixing
- Easy to forget one instance \Rightarrow bugs
- Use `reactive()` variables to reduce code duplication

- Add a reactive variable `filtered_data` that filters the players data in the same way that the outputs do
- Use the new reactive variable in the existing outputs instead of duplicating the filtering code
- Improve the plot by adding a simple theme and logged axis

```
theme_classic() +  
scale_x_log10(labels = scales::comma)
```



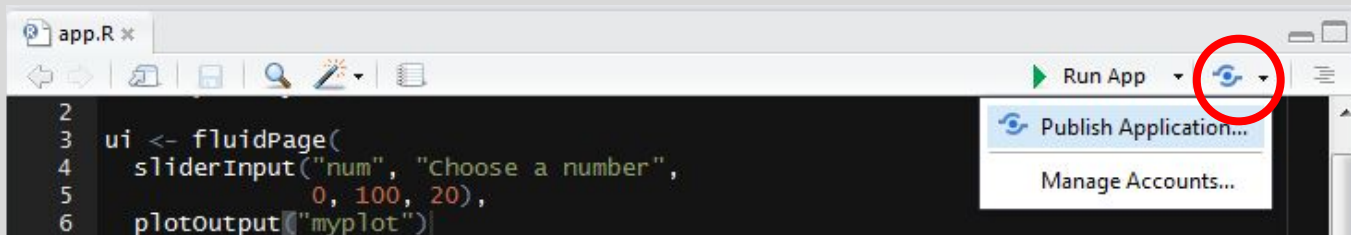
Your turn

app_06.R



Share your app: shinyapps.io

- Copy your final shiny app to a file named `app.R`
- Go to <http://www.shinyapps.io/> and make an account
- Click “Publish Application” in RStudio



- Follow instructions from RStudio
- Choose to upload `app.R` and the data file!



Your turn

PS. Shiny in Rmarkdown

- Set output: `html_document`
- Set runtime: `shiny`
- You can now use interactive inputs/outputs in Rmarkdown!

```
---  
output: html_document  
runtime: shiny  
---  
  
```${r echo=FALSE}  
sliderInput("num", "Choose a number",
 0, 100, 20)

renderPlot({
 plot(seq(input$num))
})
```,
```

- Slider inputs can be used to specify a range rather than a single value, by supplying a vector of length two to `value`. Change the `VORP` input to return a range, and filter players that are above the minimum but below the maximum.
- Dropdowns can be used to select multiple options by setting `multiple = TRUE`. Allow the user to select multiple teams to filter by.



Your turn

app_07.R



0

- Add a level 3 heading to the sidebar “Plot options”
- Add a dropdown selector with ID “variable” that allows the user to select a variable to plot, instead of always plotting players’ values. The options are: “VORP”, “Salary”, “Age”, “Height”, “Weight”. Hint: in ggplot2, you need to use `aes_string()` instead of `aes()`
- Add radio buttons with ID “plot_type” and choices “histogram”, “density” to let the user choose either a histogram or a density



plot,
Your turn

app_08.R



- Now that we can plot other variables except salary, a logged axis doesn't always make sense. Add a checkbox input with ID “log” that, when checked, causes the X axis in the plot to be logged. Use `scale_x_continuous()` for a non-logged axis.
- Add a numeric input with ID “size” that determines the font size of the plot. You can set the font size by supplying it as a parameter to `theme_classic(<size>)`.
- Add a colour input with ID “col” and an initial colour of blue. Use the colour as the `fill` aesthetic of the plot.



Your turn

app_09.R

