

Building Shiny Apps



Dean Attali
Shiny consultant

@daattali  
attalitech.com

New York R Conference
New York
June 8, 2022

Dean Attali

- Bachelor of Computer Science (University of Waterloo, Ontario)
- Past life: Software engineer (IBM, Google)
- Web developer in Silicon Valley (wish.com, tagged.com)
- MSc in Bioinformatics with Jenny Bryan (University of British Columbia)
- Since 2016: World's first Shiny consulting (AttaliTech)

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

Exercises (and slides) available at
<https://github.com/daattali/shiny-conf-nyr-2022>
or
<https://bit.ly/nyr2022shiny>

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



1. But I'm just a <insert profession>,
not a web developer!



“Building web application” sounds scary



Not with shiny - 0 knowledge of web technologies required!

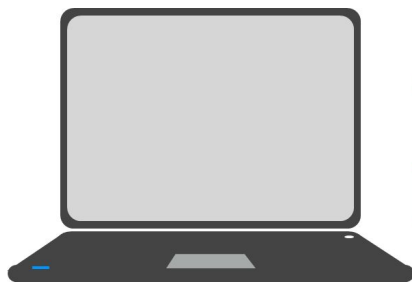
2. How is this useful?!



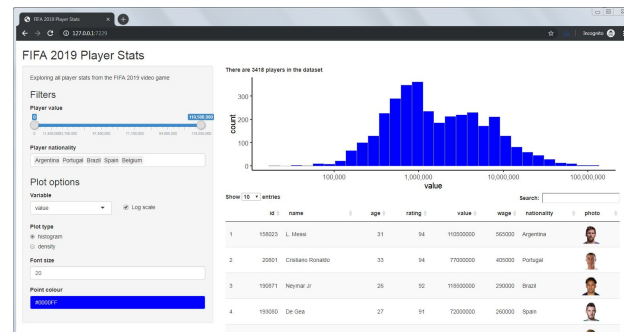
- Interactively explore data <https://daattali.com/shiny/user2017/>
- Easy interface to run an R analysis <https://daattali.com/shiny/ddpccr/>
- Dashboard/reports https://daattali.com/shiny/employee_attrition_risk/
- Just for fun <https://daattali.com/shiny/lightsout/>
- Complete websites <https://cashflowcalc.com/> <https://cranalerts.com/>
- What we'll build: Visualizing NBA 2018/19 Player Stats
<https://daattali.com/shiny/nba2018/>

Shiny Basics

What is a Shiny app?

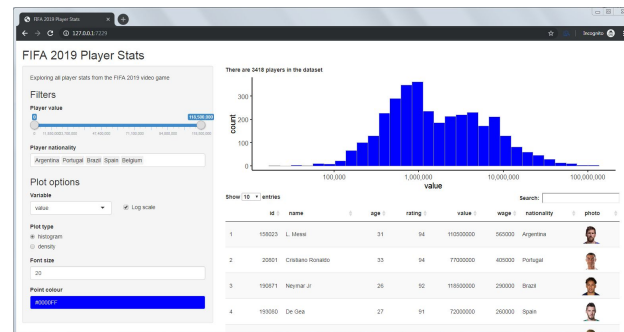


Computer
running R



Web page

What is a Shiny app?



Server code



User interface (UI)

Shiny app template

```
library(shiny)

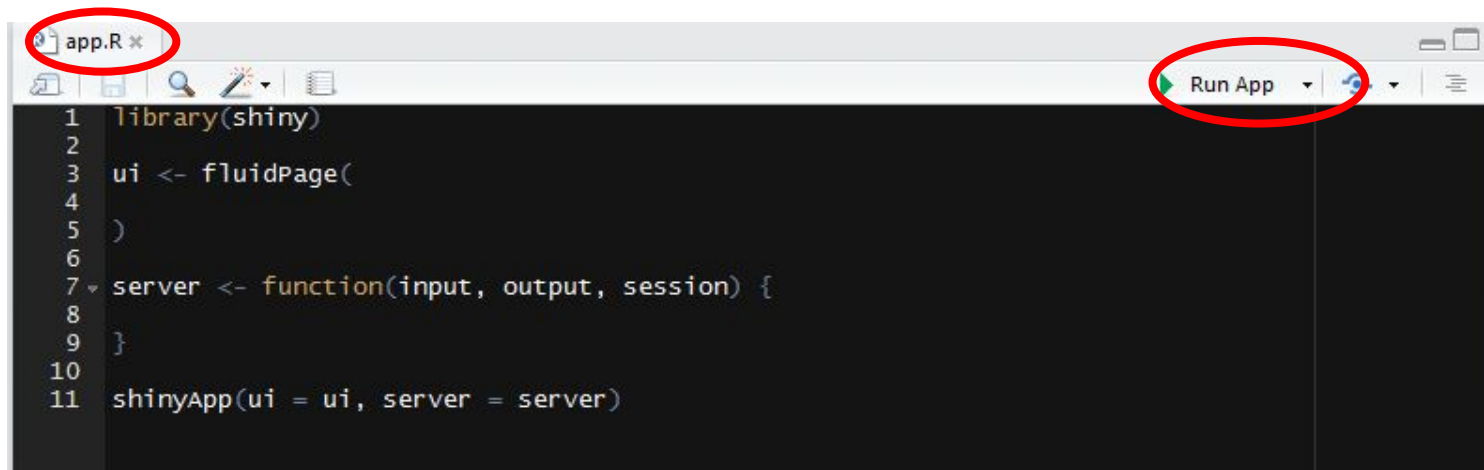
ui <- fluidPage( ... )

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

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

Run Shiny app in RStudio

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



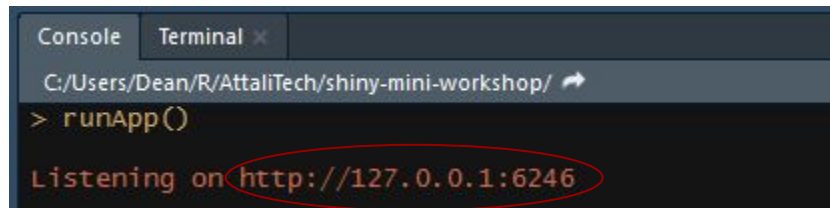
The screenshot shows the RStudio interface. The top-left pane displays a file named `app.R`, which is circled in red. The top-right pane shows the `Run App` button, also circled in red. The main editor pane contains the following R code:

```
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

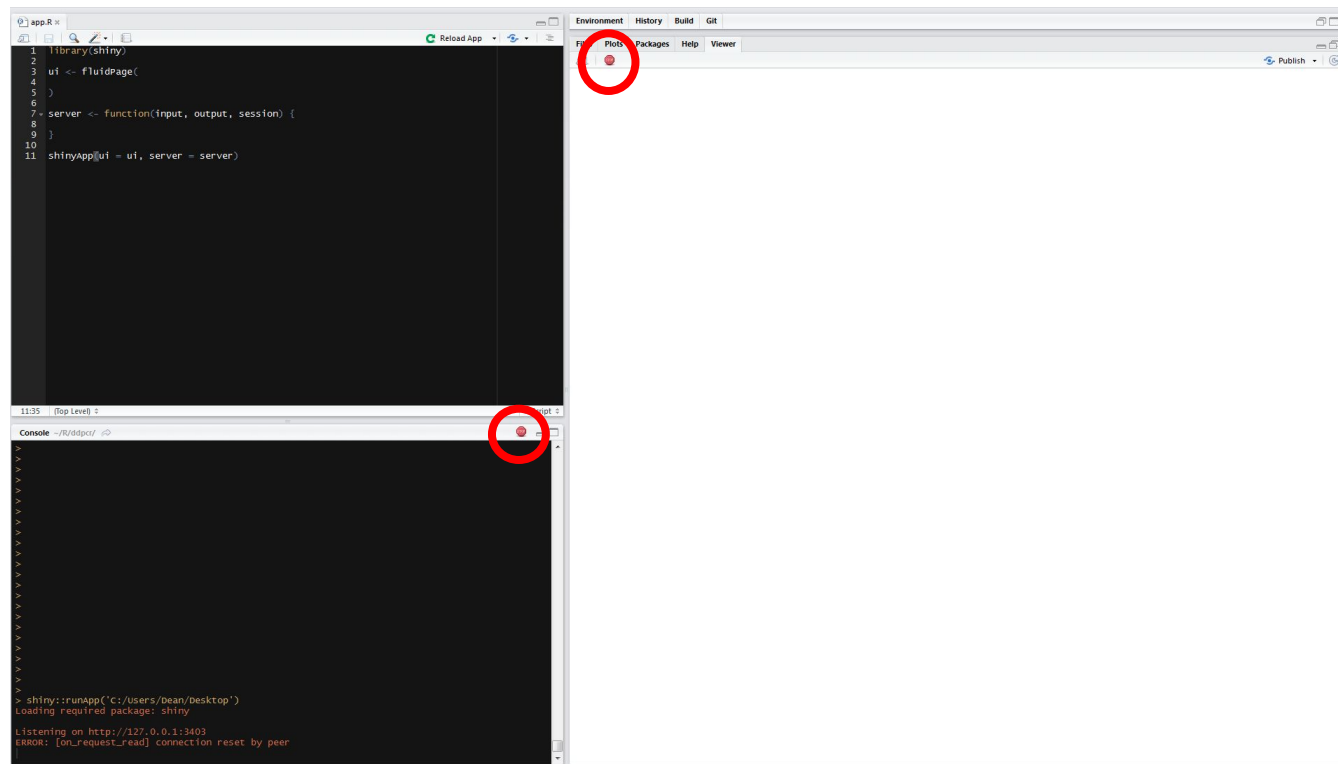


```
Console Terminal x  
C:/Users/Dean/R/AttaliTech/shiny-mini-workshop/ ↗  
> runApp()  
Listening on http://127.0.0.1:6246
```

A screenshot of an R console window. The window has two tabs: 'Console' and 'Terminal'. The 'Console' tab is active. The path 'C:/Users/Dean/R/AttaliTech/shiny-mini-workshop/' is shown above the prompt. The command '> runApp()' has been entered. The output 'Listening on http://127.0.0.1:6246' is displayed, with the URL 'http://127.0.0.1:6246' circled in red. A red arrow points from this URL down to the text below.

Console tells you URL for app - can open in browser

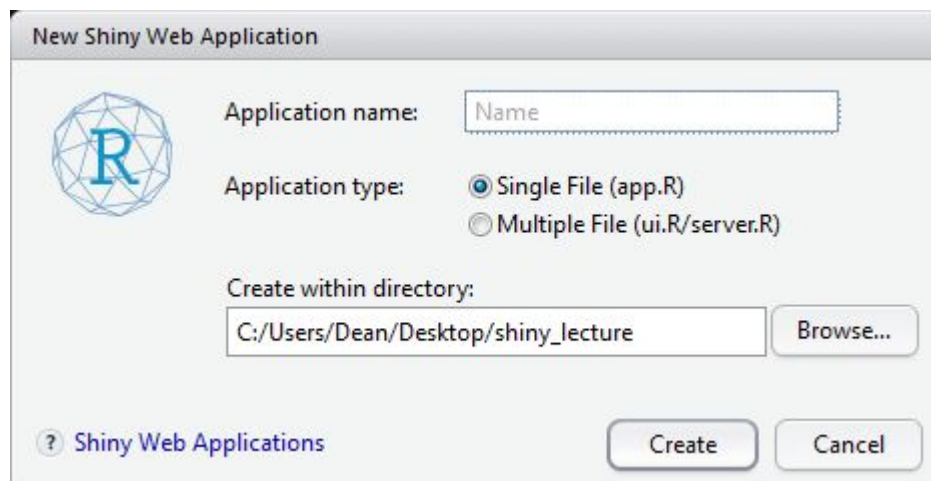
Stop Shiny app in RStudio



Press
Escape
or click
the *Stop*
icon

A little cheat..

File > New File > Shiny Web App...



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

- Load the dataset after loading {shiny}, before defining the UI:

```
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?



nba/app_00.R



User Interface (UI)

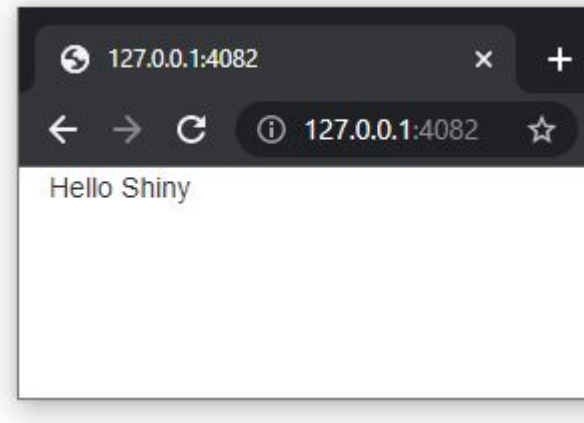
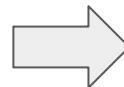
Add text as argument to `fluidPage()`

```
library(shiny)

ui <- fluidPage(
  "Hello Shiny"
)

server <- function(input, output) {}

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



`fluidPage()` accepts arbitrary number of arguments

Formatted text

`h1 ()` **Primary header**

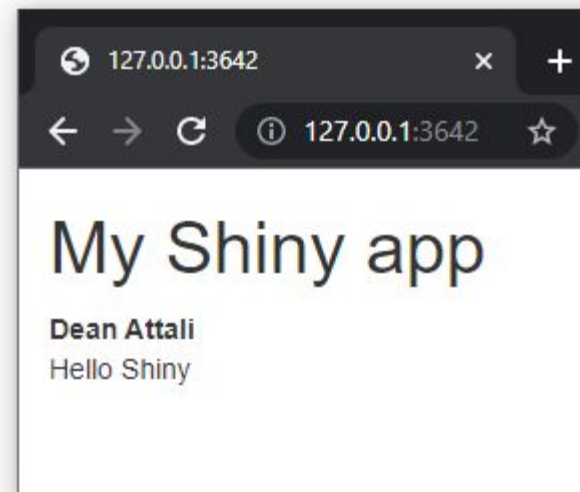
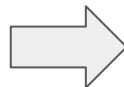
`h2 ()` Secondary header

`strong ()` **Bold**

`em ()` *Italicized (emphasized)*

`br ()` Line break

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



Watch out for commas!

UI functions are simply HTML wrappers

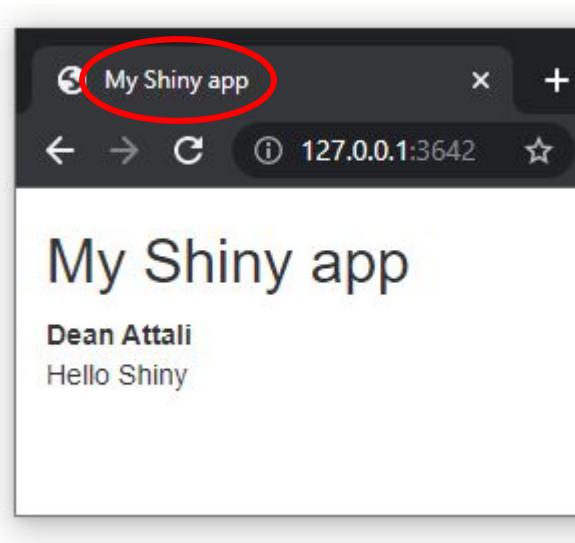
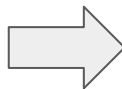
```
> print(ui)
<div class="container-fluid">
  <h1>My Shiny app</h1>
  <strong>Dean Attali</strong>
  <br/>
  Hello Shiny
</div>
```

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"     "eventsourcing" "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 Shiny"  
)
```



Get the app to look like the one on the next page:

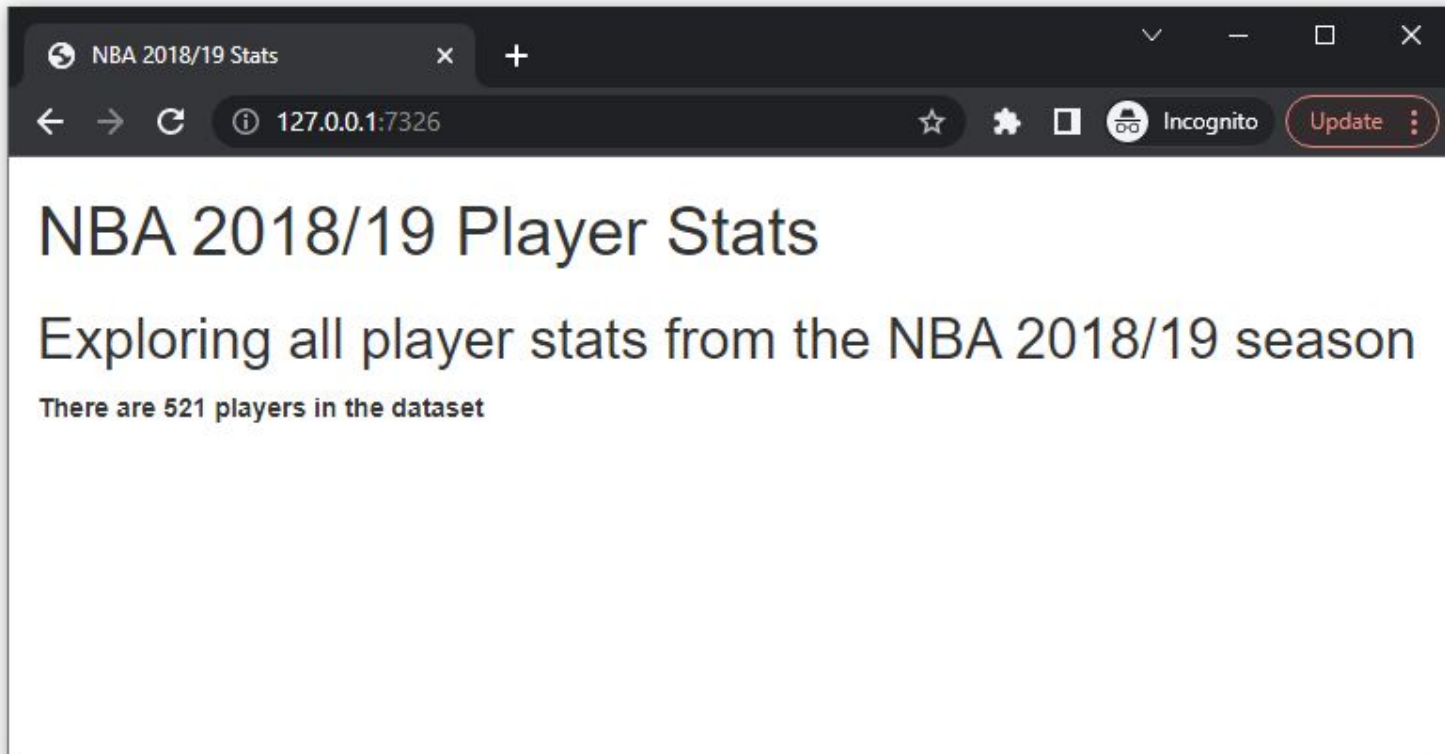
- First get the app to run, it's currently broken
- First line is the title (header)
- Second line is subtitle (subheader)
- The rest of the text that describes the data size should be bolded
- Add a title to the browser tab that is different than the title on the page: “NBA 2018/19 Stats”. *Hint: read docs for*



fluidPage()

nba/app_01.R





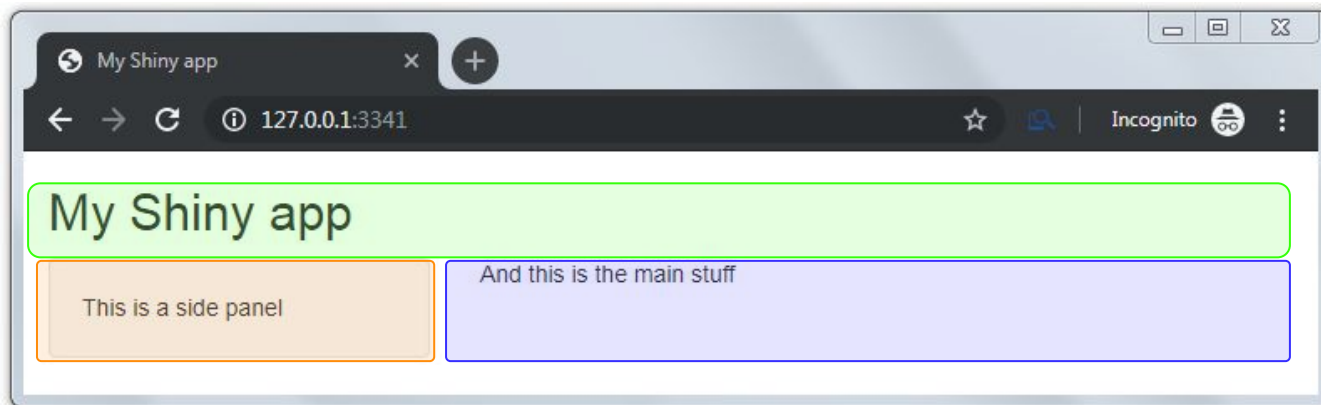
nba/app_01.R



Layouts

- By default, all elements stack up one after the other
- Layout adds control over positioning
- Three important to know: sidebar layout, rows + columns, tabs

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



UI functions are STILL just HTML

```
> 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>
```


- 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”



nba/app_02.R

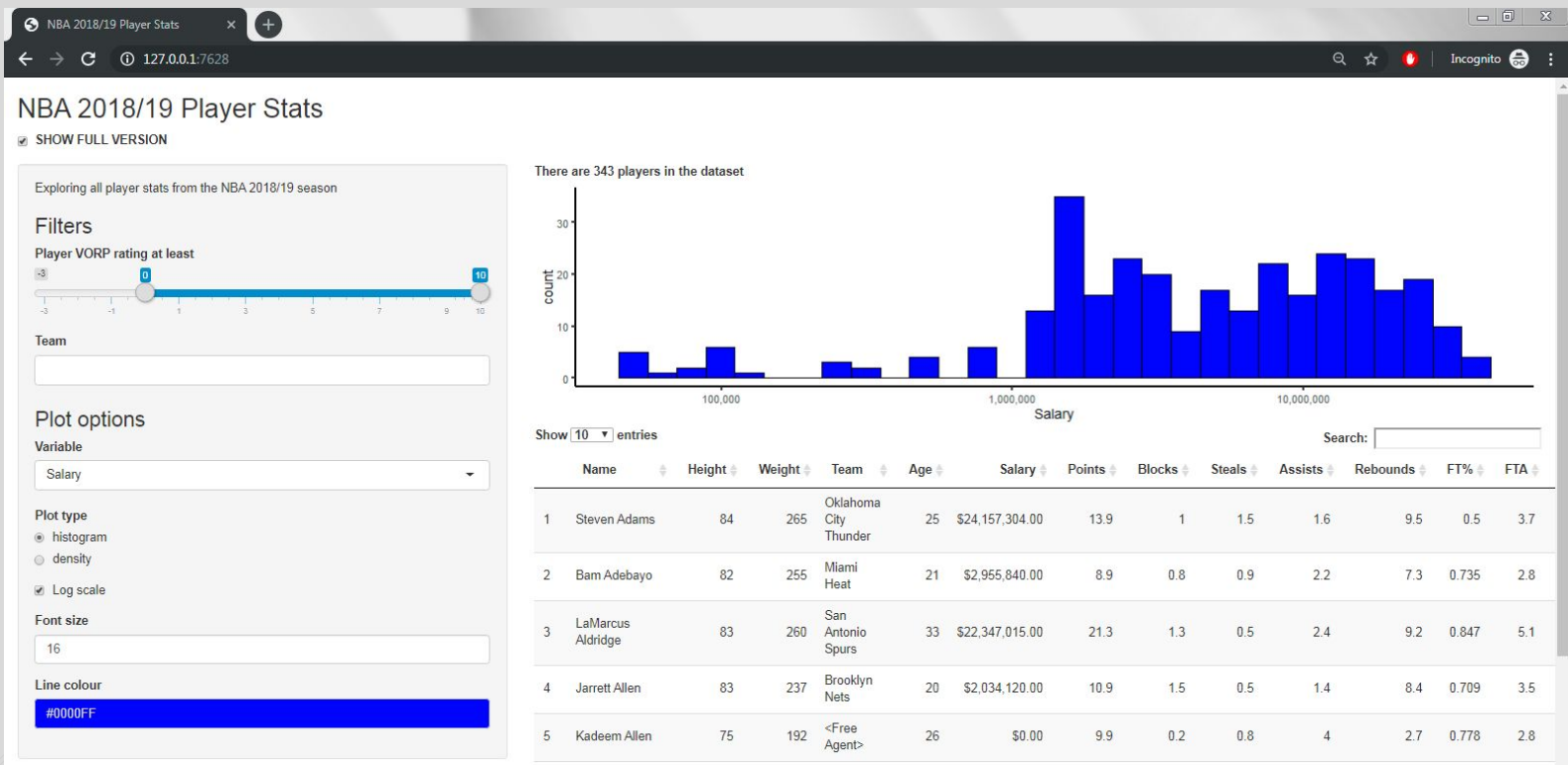


Inputs and Outputs

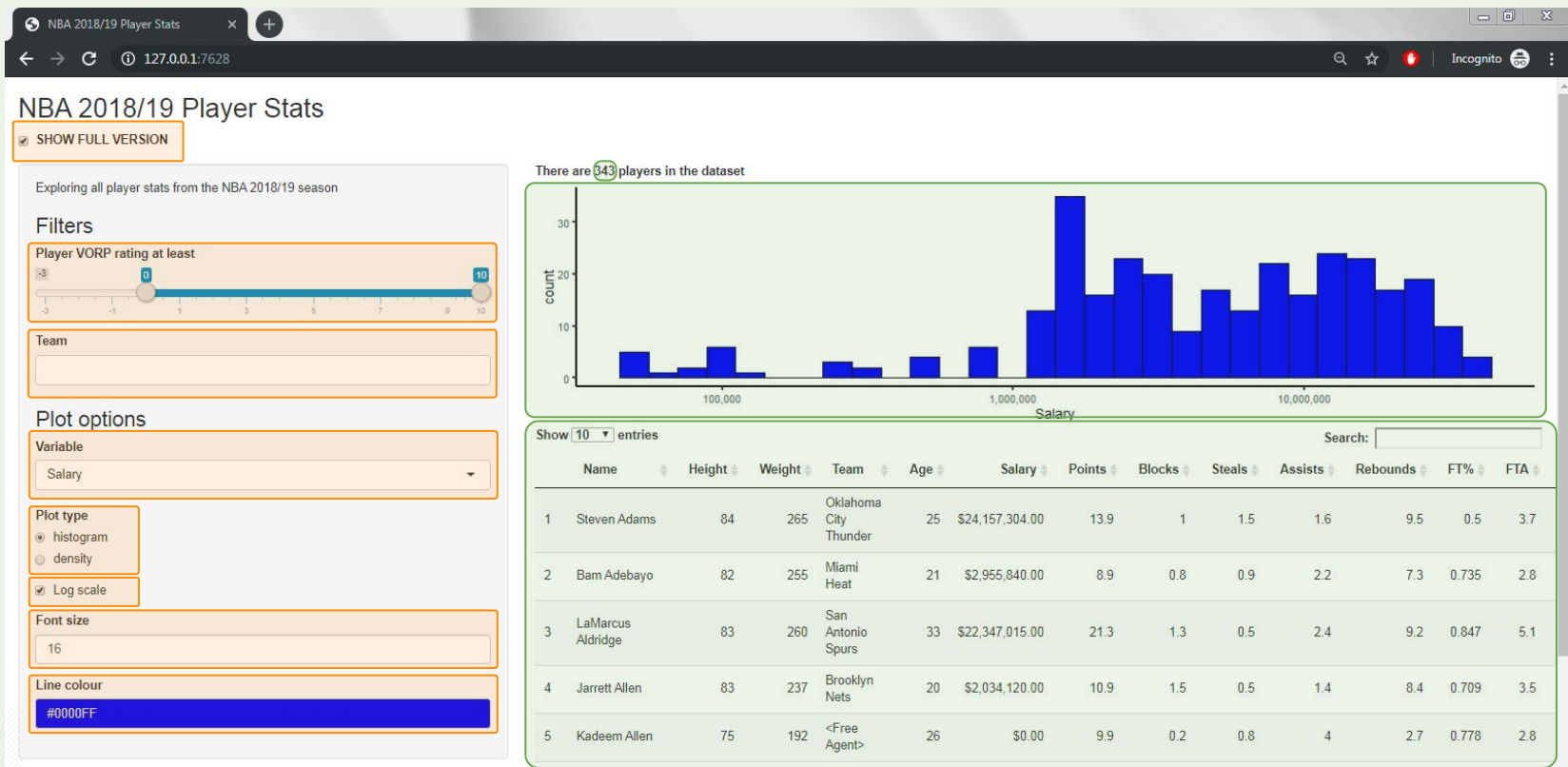
- For interactivity, app needs inputs and outputs
- **Inputs** - things user can toggle
- **Output** - R objects user can see, often depend on inputs

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

How many inputs? How many outputs?



8 inputs, 3 outputs



Buttons

`actionButton()`
`submitButton()`

Date range to

`dateRangeInput()`

Radio buttons

☒ Choice 1
☐ Choice 2
☐ Choice 3

`radioButtons()`

Single checkbox

☒ Choice A

`checkboxInput()`

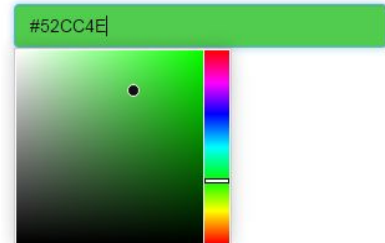
Checkbox group

☒ Choice 1
☐ Choice 2
☐ Choice 3

`checkboxGroupInput()`

Date input

`dateInput()`

Colour input

`colourpicker::colourInput()`

File input

No file chosen

`fileInput()`

Numeric input

`numericInput()`

Password Input

`passwordInput()`

Sliders

`sliderInput()`

Text input

`textInput()`

Select box

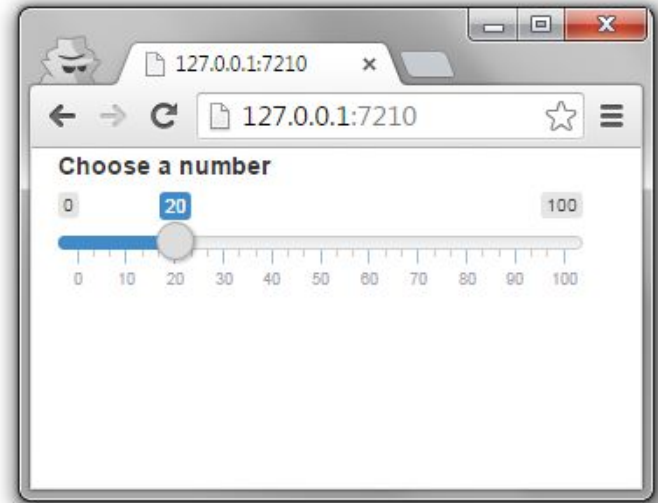
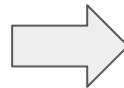
`selectInput()`

```
library(shiny)

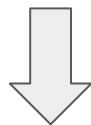
ui <- fluidPage(
  sliderInput(
    inputId = "num",
    label = "Choose a number",
    min = 0, max = 100,
    value = 20
  )
)

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>
```


Input id Label to display Input-specific arguments

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

```
*Input(inputId, label, ...)
```

What arguments can I pass to an input function?

```
?sliderInput
```



ID must be unique!

- 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.



nba/app_03.R



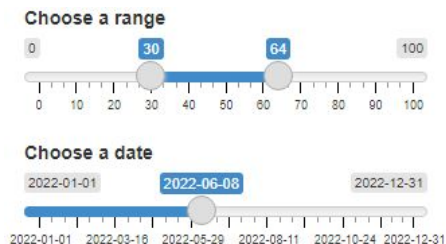
- Can use `label = NULL` to save space
 - Paired well with `placeholder` argument

```
textInput("name", label = NULL, "",
          placeholder = "Enter your name")
```

Enter your name

- Always read inputs parameters and documentation!

Sliders can have a range
of numbers or dates



Dropdowns can have
sub-sections

The figure shows a Shiny dropdown menu titled "Choose a state:". The top input box contains "NY". Below the box, the options are grouped into two sections: "East Coast" and "West Coast". Under "East Coast", the options are "NY", "NJ", and "CT". Under "West Coast", the options are "WA" and "OR".

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>	a plot
<code>tableOutput()</code>	a table
<code>textOutput()</code>	a text
<code>uiOutput()</code>	Shiny UI

Output name Output-specific arguments

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

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

What arguments can I pass to an output function?

```
?plotOutput
```



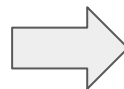
ID must be unique and not clash with any input IDs!

```
library(shiny)

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

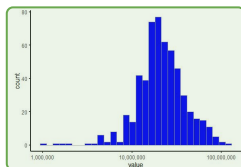
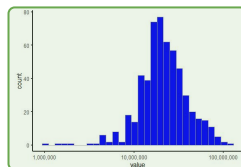
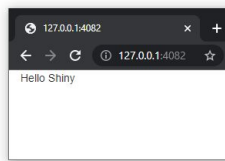
server <- function(input, output) {}

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!



nba/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>`

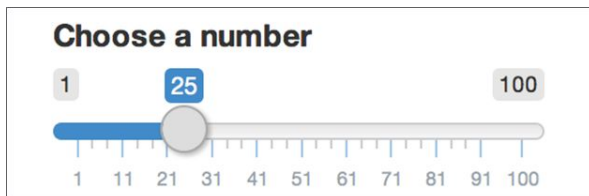
```
server <- function(input, output) {  
  
  output$myplot <- renderPlot({  
    plot(rnorm(input$num))  
  })  
  # in UI: plotOutput("myplot")  
}
```

Building outputs

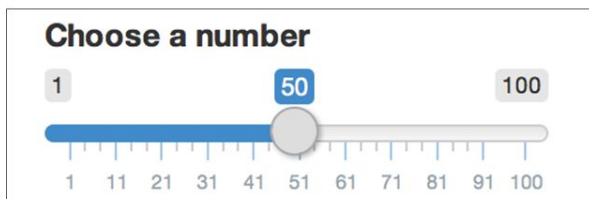
3 - Access input values with `input$id`

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

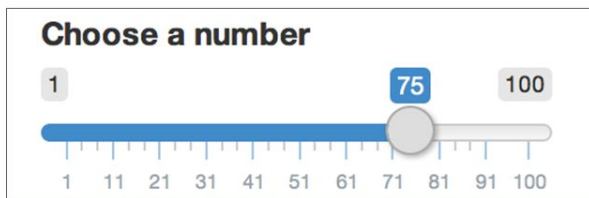
Using input\$



`input$num` returns 25



`input$num` returns 50



`input$num` returns 75

This app should show the product of a number and 5, but it doesn't work. Try to spot out the error without running it, then fix it.

```
library(shiny)
ui <- fluidPage(
  sliderInput("x", label = "If x is",
             min = 1, max = 50, value = 30),
  "then x times 5 is",
  textOutput("product")
)

server <- function(input, output, session) {
  output$product <- renderText({
    x * 5
  })
}

shinyApp(ui, server)
```



Solution on next page

The 'x' inside the output should be 'input\$x'

```
library(shiny)

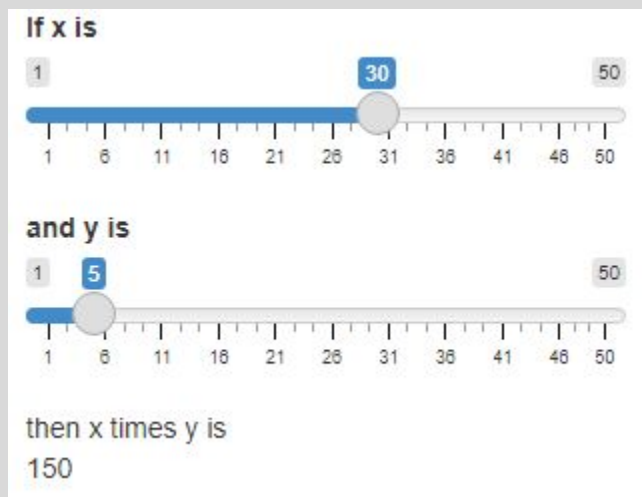
ui <- fluidPage(
  sliderInput("x", label = "If x is", min = 1, max = 50, value = 30),
  "then x times 5 is",
  textOutput("product")
)

server <- function(input, output, session) {
  output$product <- renderText({
    input$x * 5
  })
}

shinyApp(ui, server)
```



Add another input 'y' that will be the multiplier, to result in this app



Solution on next page



```
library(shiny)

ui <- fluidPage(
  sliderInput("x", label = "If x is", min = 1, max = 50, value = 30),
  sliderInput("y", label = "and y is", min = 1, max = 50, value = 5),
  "then x times y is",
  textOutput("product")
)

server <- function(input, output, session) {
  output$product <- renderText({
    input$x * input$y
  })
}

shinyApp(ui, server)
```



- The app is meant to show three outputs (you can see them in the UI), but none of them are working. Fix the app so that all three outputs render correctly.
- There is also one “silent bug” - the app appears to work, but something isn’t quite right. *Hint: change the input and see if the plot updates.*



diamonds/outputs.R



- 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*



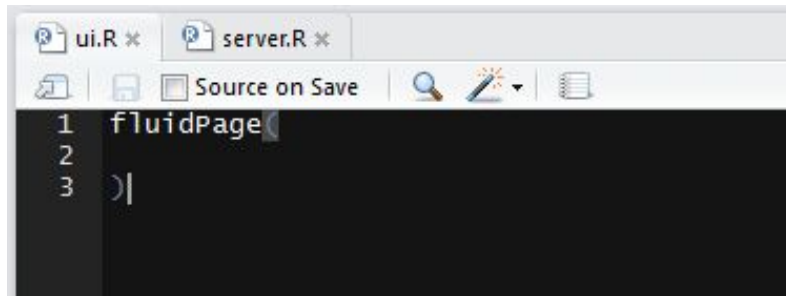
nba/app_05.R



Back to Basics

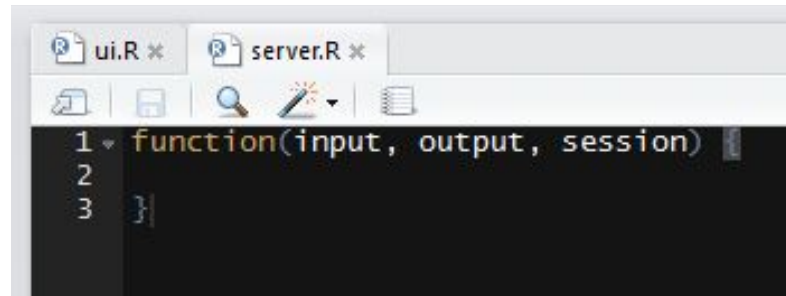
Run Shiny app as multiple files

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



A screenshot of the RStudio editor showing the `ui.R` file. The file contains three lines of code: `1 fluidPage`, `2` (blank line), and `3)|`. The RStudio interface includes a toolbar with icons for file operations and a 'Source on Save' button.

```
1 fluidPage
2
3 )|
```



A screenshot of the RStudio editor showing the `server.R` file. The file contains three lines of code: `1 function(input, output, session)`, `2` (blank line), and `3 }`. The RStudio interface includes a toolbar with icons for file operations and a 'Source on Save' button.

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

Good for complex Shiny apps, separates view vs logic



Do not call `shinyApp(...)`

Run Shiny app as multiple files

- Optionally “global.R” file
- Loads objects in global env
- Define objects used in both ui and server, load libraries used in both

Order of Execution

- global.R first, then ui.R
 - Both only run **once** when app initializes for first user
- Server runs once per user
 - Every time page loads (including page refresh)
- Little known fact: UI is cached
 - Wrap in `function(req) {}` to escape UI caching

Scope

- Anything defined inside server is specific to that session
 - This includes input, output, any reactives and variables
- If an object needs to be shared with all users, define it outside of server (or in global.R)
 - Large datasets or utility functions

```
library(shiny)
library(magrittr)
```

ui.R

```
time_1 <- Sys.time() %>% format_custom()
Sys.sleep(1)
```

```
ui <- fluidPage(
  h4(paste("Time 1:", time_1)),
  h4(textOutput("time_2")),
  h4(paste("Time 3:", Sys.time() %>% format_custom())),
  h4(paste("Time 4:", time_4))
)
```

```
server <- function(input, output) {
  output$time_2 <- renderText({
    paste("Time 2:", Sys.time() %>% format_custom())
  })
}
```

server.R

```
format_custom <- function(x){
  format(x, "%H:%M:%OS2")
}

time_4 <- Sys.time() %>% format_custom()
Sys.sleep(1)
```

global.R

1. What error will running the app using the "Run app" button or with the `runApp()` function yield? How would you resolve this error? *Hint: The pipe operator (`%>%`) is part of the {magrittr} package.*
2. Without running the app, predict in what order the timestamps will show up.
3. If you were to open the app in a new tab, which timestamps will be different?
4. Suppose you deploy this app and share the URL with a friend, and they immediately visit it on their own computer. Which timestamps will be different than what you see on your screen?

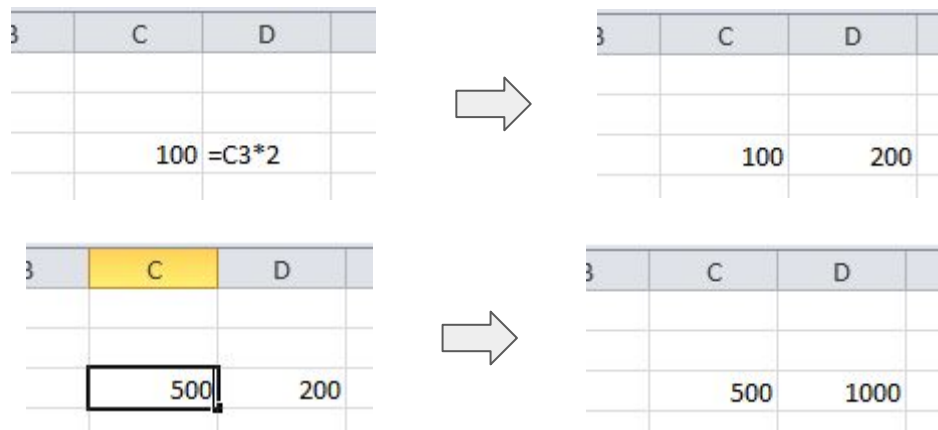


Reactivity

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



- 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  
  })  
}
```



Custom `reactive()` 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

```
x <- reactive({  
  input$num1 + 5  
})  
y <- reactive({  
  x() + input$num2  
})
```

When does the value of `y` get updated?

1. When the user changes the value of `num1` input only
2. When the user changes the value of `num2` input only
3. When the user changes the value of either `num1` or `num2`
4. Only after the user changes the values of both `num1` and `num2`



- 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)
```



nba/app_06.R



- 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.



nba/app_07.R



- 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.



nba/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.

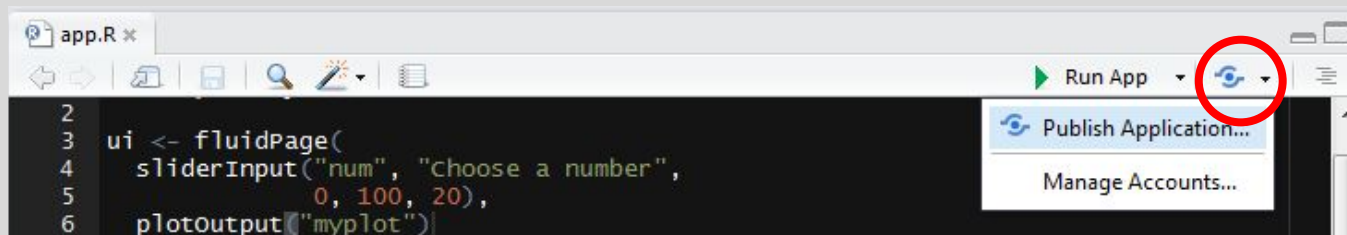


nba/app_09.R



Share your app: shinyapps.io

- Copy your final shiny app to a file named `app.R`
- Go to <https://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!**

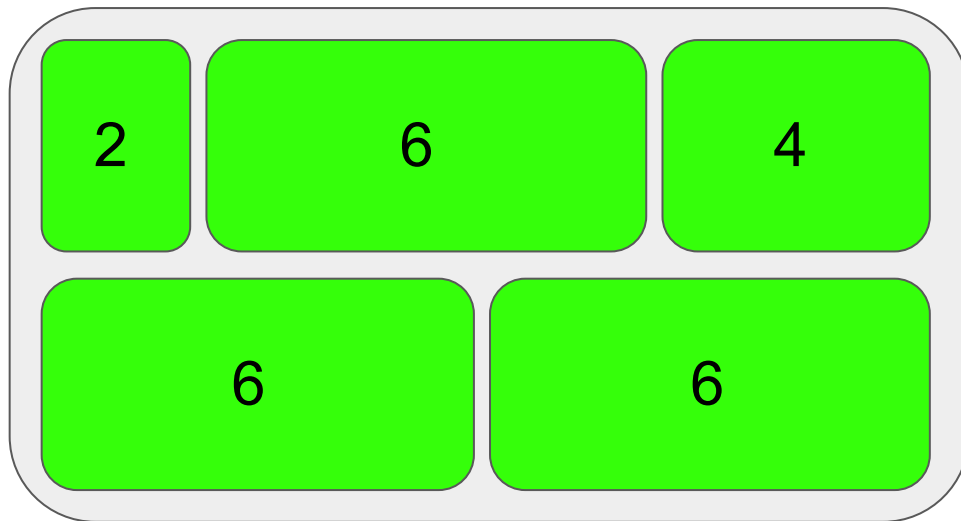


Intermediate UI

Grid System

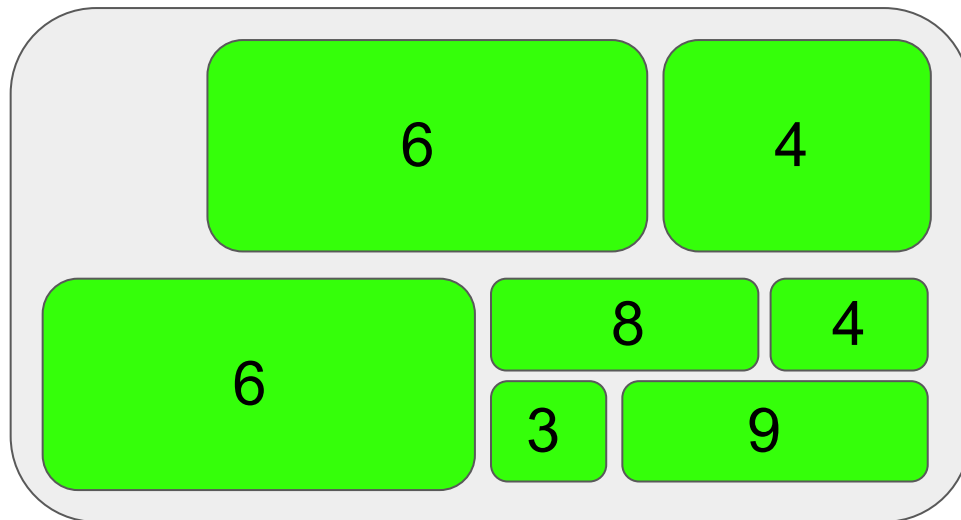
- `fluidRow()` and `column()`
- Each row divided into 12
- Column must be inside row, row can only contain column

```
fluidRow(  
  column(2, "2"),  
  column(6, "6"),  
  column(4, "4")  
) ,  
fluidRow(  
  column(6, "6"),  
  column(6, "6")  
)
```



- Can use an offset
- Rows can be inside other rows

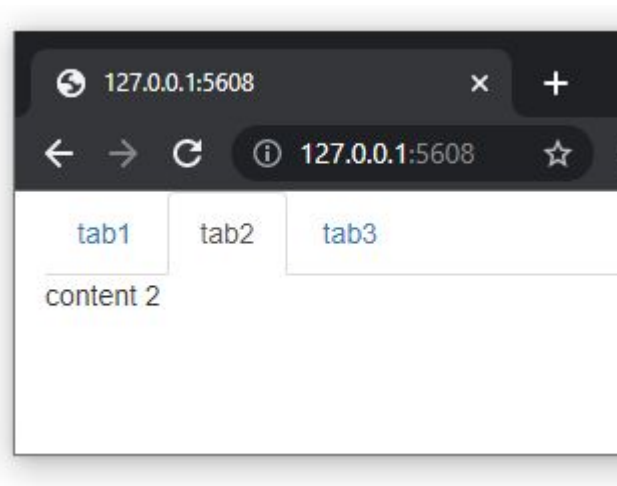
```
fluidRow(  
  column(  
    offset = 2, 6, "6"),  
    column(4, "4")  
  ),  
fluidRow(  
  column(6, "6"),  
  column(6,  
    fluidRow(  
      column(8, "8"),  
      column(4, "4")  
    ), fluidRow(  
      column(3, "3"),  
      column(9, "9"))))
```



Tabs

- `tabsetPanel()` with a list of `tabPanel()`

```
tabsetPanel(  
  tabPanel("tab1", "content 1"),  
  tabPanel("tab2", "content 2"),  
  tabPanel("tab3", "content 3")  
)
```



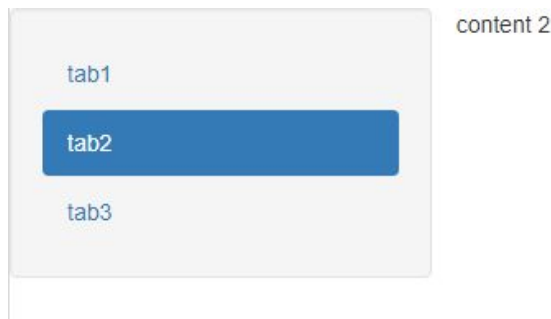
- **Protip:** give an ID to `tabsetPanel()` → `input$id` will tell you which tab is selected

Tabs

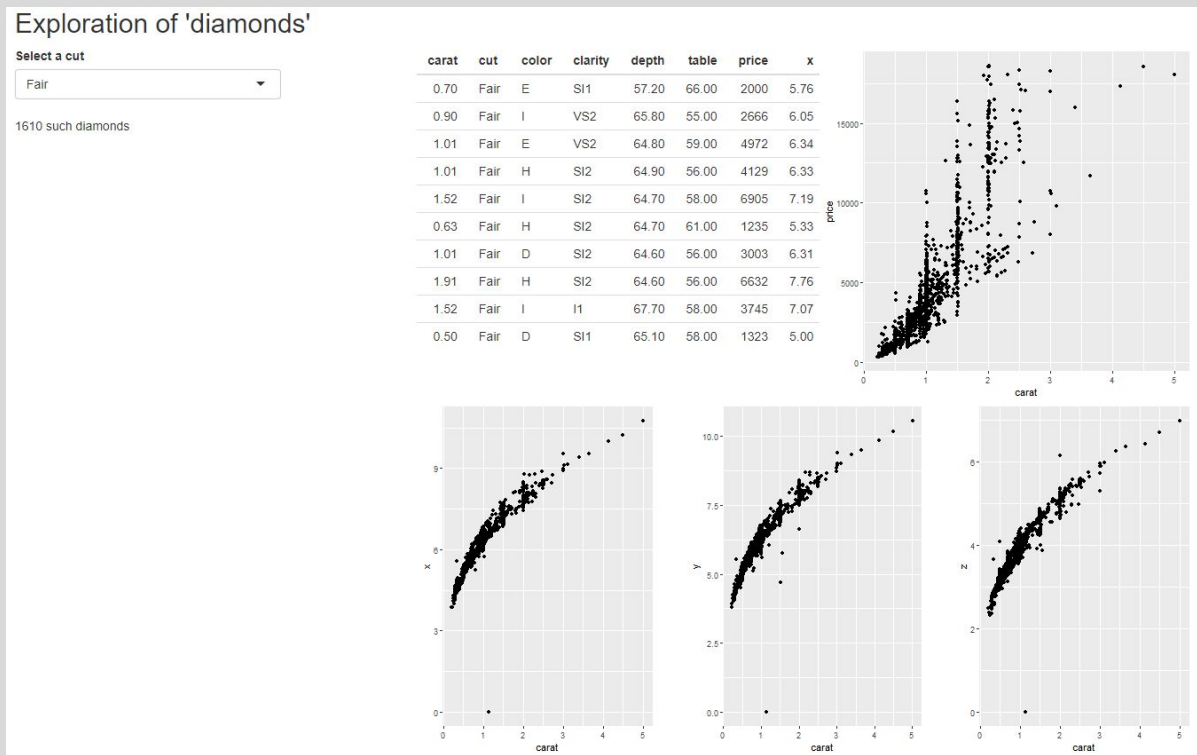
- Use `type = "pills"` for a different look



- Use `navlistPanel()` for vertical tabs



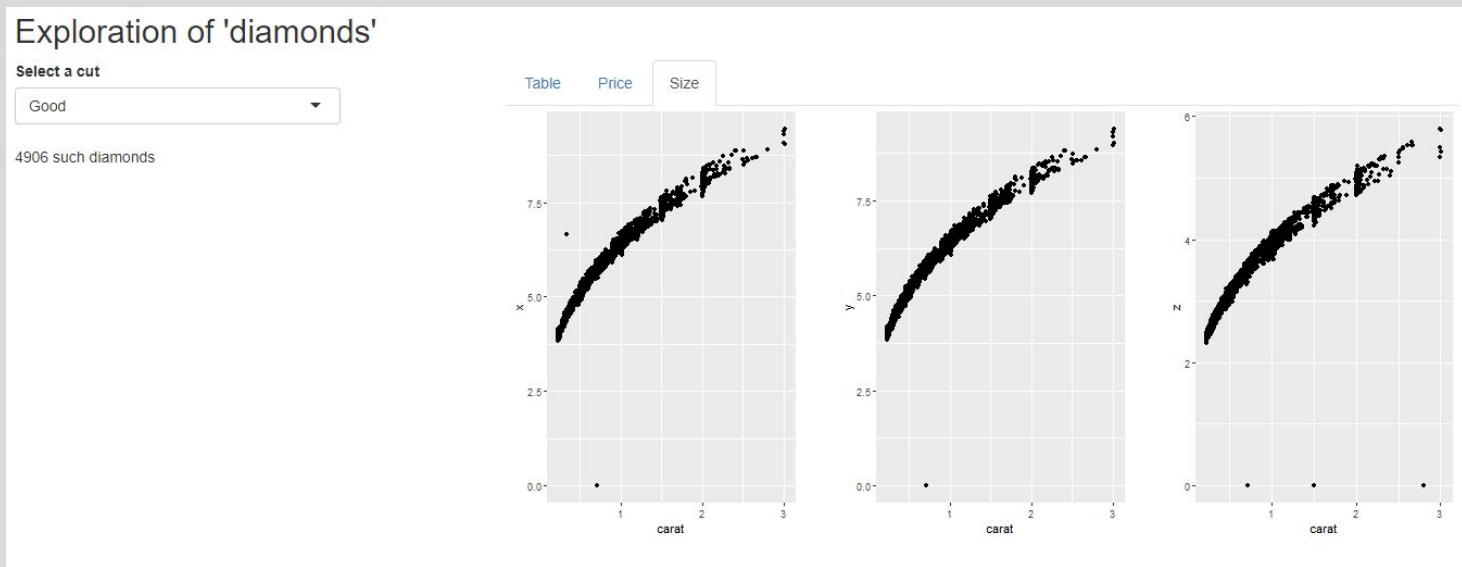
- Modify the app's UI to match this image:



diamonds/grid.R



- Modify the app's UI such that the outputs are divided into 3 tabs, to match this image:



diamonds/tabs.R



Verbatim Text

- Regular `textOutput()` places text in regular HTML
- Use `verbatimTextOutput()` to place text in a fixed-width (code-like) container

```
ui <- fluidPage(  
  textOutput("text1"),  
  verbatimTextOutput("text2")  
)  
  
server <- function(input, output, session) {  
  output$text1 <- renderText("new\nline")  
  output$text2 <- renderText("new\nline")  
}  
  
shinyApp(ui, server)
```

new line

new
line

uiOutput() : Dynamic UI

- An output to render UI
- Can create inputs (or any UI) dynamically
- Useful when inputs need to change

```
fluidPage(  
  sliderInput(  
    "min", "Minimum",  
    min = 0, max = 99, value = 0  
  ),  
  uiOutput("max_input")  
)
```

ui.R

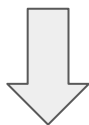
```
function(input, output) {  
  output$max_input <- renderUI({  
    sliderInput(  
      "max", "Maximum",  
      min = input$min,  
      max = 100, value = 100  
    )  
  })  
}
```

server.R

Returning multiple elements

- `div(...)` : a “container” for HTML elements, creates a new element
- `tagList(...)` : a list of HTML elements

```
div("some text", plotOutput("plot"))
```



```
<div>
  some text
  <div id="plot"
class="shiny-plot-output"
style="width:100%;height:400px;"></div>
</div>
```

```
tagList("some text",
plotOutput("plot"))
```



```
some text
<div id="plot" class="shiny-plot-output"
style="width:100%;height:400px;"></div>
```

fileInput()

- Allows user to upload files **from their computer**

```
fileInput(  
  id = "file",  
  label = "Choose a file",  
  multiple = TRUE, # optional  
  accept = ".csv" # optional  
)
```

- In server, `input$id` is `data.frame`
 - One row per file
 - Columns: name, size, type, **datapath**
- For security reasons, you cannot know the real path

- Currently the app only works with the `mtcars` dataset
- Add a file input in the sidebar to choose one file
- Create a `full_data()` reactive that contains the data
- Modify the server code to use this data instead of `mtcars`



You will see some error messages until a file is uploaded.
That's fine for now, we'll fix that later.



`mtcars/fileinput.R`



Conditional UI

- Use `conditionalPanel()` to conditionally show/hide UI
- First argument is `condition` – a JavaScript expression
 - Similar to R but use "." instead of "\$" for inputs
 - `condition = "input.num > 5"`
 - `condition = "input.team == Toronto Raptors"`
 - `condition = "input.show_plot"`
or `"input.show_plot == true"`
- Remaining arguments are UI

- A checkbox has been added - “Choose my own file”
- Modify `full_data()` to either use the user’s file or `mtcars`, depending on whether the checkbox is checked
- Use a conditional panel to only show the file input when it’s needed



mtcars/conditional.R



Intermediate Reactivity

Reactives

- Help reduce code duplication
- Help performance by being **lazy**
 - Only run when needed
 - If nobody calls a reactive or it's output is hidden, no run
- Help performance by **caching value**
 - Only re-run when dependencies change

Observe vs Reactive

<code>reactive()</code>	<code>observe()</code>
Lazy	Eager
Used for value	Used for side effect
Caches value	Nothing returned

Both are reactive contexts that re-run if any of their (reactive) dependencies are modified

Suppressing reactivity

- Reactive block gets re-executed when **any** dependencies change
- Use `isolate()` around a reactive value to suppress this
- Can isolate a single value, an entire line, or a code block

- A. `observe({ input$x + input$y })`
- B. `observe({ isolate(input$x) + input$y })`
- C. `observe({ input$x + isolate(input$y) })`
- D. `observe({ isolate(input$x + input$y) })`

When do each of these observers trigger?



Suppressing reactivity - buttons

- `isolate()` commonly used with buttons if don't want every input change to cause an update
- `actionButton(id, label)`
- For example, plot with several parameters, want to wait until button is pressed before updating
- Button input value is how many times it was pressed
 - Rarely used for its value

Going back to the dataset exploration app, it had many errors before a dataset is loaded.

See if you can figure out how to make all the errors disappear.



mtcars/req.R



`req()` - Checking Conditions

- Checks for “truthy” rather than TRUE
- Rather than show error (like `stop()`), make output blank
- `req(input$text)` Ensure user provided value for text input
- `req(input$button)` Ensure button was pressed at least once
- `req(x %% 2 == 0)` Ensure x is an even number
- `req(FALSE)` Cancel the current reactive/observer/output

This is a counter app. Increment/decrement should change the value by 1, reset should set it back to 0. See if you can get it to work.

```
library(shiny)

ui <- fluidPage(
  actionButton("increment", "Increment"),
  actionButton("decrement", "Decrement"),
  actionButton("reset", "Reset"),
  textOutput("value")
)

server <- function(input, output, session) {
  output$value <- renderText({
    0
  })
}

shinyApp(ui, server)
```



Solution on next page

Not possible with only using simple reactivities



reactiveVal()

- `reactive()` Major drawback: only one entry point → cannot modify on-demand
- `reactiveVal()` A reactive that we control manually
- Syntax a bit awkward:
 - Create `val <- reactiveVal(0)`
 - Access `val()`
 - Modify `val(5)`
- Accessing reactive creates a dependency, modifying doesn't

```
library(shiny)
ui <- fluidPage(
  actionButton("btn", "btn"),
  numericInput("num", "num", 5)
)

server <- function(input, output, session) {
  x <- reactiveVal(0)

  observe({
    newval <- input$num * 2    # A
    x(newval)                  # B
  })

  observe({
    input$btn                  # C
    val <- x()                  # D
    print(val)
  })
}

shinyApp(ui, server)
```

What would happen if line A is wrapped in `isolate()`?

What about line B? And C? And D?



- Useful for storing values that can't be determined from the current state of all inputs

```
observe({  
  req(input$add)  
  isolate( total( total() + input$x ) )  
})
```

- Useful as a variable that keeps track of an event

```
observe({ req(input$editMode); mode("edit") })  
observe({ req(input$previewMode); mode("preview") })  
output$page <- renderUI({  
  if (mode() == "edit") {  
    ...  
  } else if (mode() == "preview") {  
    ...  
  }  
})
```

Go back to the counter app and make it work using `reactiveVal()`

```
library(shiny)

ui <- fluidPage(
  actionButton("increment", "Increment"),
  actionButton("decrement", "Decrement"),
  actionButton("reset", "Reset"),
  textOutput("value")
)

server <- function(input, output, session) {
  value <- reactiveVal(0)
  output$value <- renderText({
    value()
  })
}

shinyApp(ui, server)
```



Solution on next page

```
server <- function(input, output, session) {  
  value <- reactiveVal(0)  
  
  observe({  
    req(input$increment)  
    isolate( value( value() + 1 ) )  
  })  
  
  observe({  
    req(input$decrement)  
    isolate( value( value() - 1 ) )  
  })  
  
  observe({  
    req(input$reset)  
    isolate( value(0) )  
  })  
  
  output$value <- renderText({  
    value()  
  })  
}
```



```
value <- reactiveVal(0)

observe({
  req(input$increment)
  isolate( value( value() + 1 ) )
})

observe({
  req(input$decrement)
  value( value() - 1 )
})

observe({
  req(input$reset)
  isolate( value(0) )
})

output$value <- renderText({
  value()
})
}
```

What would happen if the
`isolate()` in the decrement
is removed?



<code>reactive()</code>	<code>reactiveVal()</code>
Updates automatically	Updates manually
Value defined in one place	Value defined in multiple places
Lazy	Not lazy
Easier to debug (can always know the value based on app state)	More difficult to debug
Preferred usage when possible	Use only when <code>reactive()</code> won't work

This app works correctly, but it's more complex than it needs to be.
Refactor it to simplify the code.

```
fluidPage(  
  "Pythagorean theorem:  $a^2 + b^2 = c^2$ ",  
  numericInput("a", "a", 3),  
  numericInput("b", "b", 4),  
  "c:",  
  textOutput("c")  
)
```

ui.R

```
a2 <- reactiveVal(0)  
b2 <- reactiveVal(0)  
c2 <- reactiveVal(0)
```

server.R

```
observe({  
  a2(input$a ** 2)  
})  
  
observe({  
  b2(input$b ** 2)  
})  
  
observe({  
  c2(a2() + b2())  
})  
  
output$c <- renderText({  
  sqrt(c2())  
})
```



Solution on next page



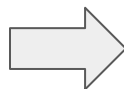
```
server <- function(input, output, session) {  
  a2 <- reactive({  
    input$a ** 2  
  })  
  
  b2 <- reactive({  
    input$b ** 2  
  })  
  
  c2 <- reactive({  
    a2() + b2()  
  })  
  
  output$c <- renderText({  
    sqrt(c2())  
  })  
}
```



Controlling reactivity - `observeEvent()`

- Often we want observer to only depend on one thing and ignore everything else
- `observeEvent()` is functionally identical but easier to read and write

```
observe({  
  req(input$multiply)  
  isolate( value( input$x * input$y ) )  
})
```



```
observeEvent(input$multiply, {  
  value( input$x * input$y )  
})
```

Go back to the counter app and use `observeEvent()` instead

```
fluidPage(  
  actionButton("increment", "Increment"),  
  actionButton("decrement", "Decrement"),  
  actionButton("reset", "Reset"),  
  textOutput("value")  
)
```

ui.R

```
function(input, output, session) {  
  value <- reactiveVal(0)  
  
  observe({  
    req(input$increment)  
    isolate( value( value() + 1 ) )  
  })  
  
  observe({  
    req(input$decrement)  
    isolate( value( value() - 1 ) )  
  })  
  
  observe({  
    req(input$reset)  
    isolate( value(0) )  
  })  
  
  output$value <- renderText({  
    value()  
  })  
}
```

server.R

Solution on next page



```
server <- function(input, output, session) {  
  value <- reactiveVal(0)  
  
  observeEvent(input$increment, {  
    value( value() + 1 )  
  })  
  
  observeEvent(input$decrement, {  
    value( value() - 1 )  
  })  
  
  observeEvent(input$reset, {  
    value(0)  
  })  
  
  output$value <- renderText({  
    value()  
  })  
}
```



This app is meant to only show a new set of random numbers when the button is pressed, but it also updates when the input changes. Fix it.

```
library(shiny)

ui <- fluidPage(
  numericInput("n", "Number of observations", 5),
  actionButton("recalculate", "Recalculate"),
  verbatimTextOutput("numbers")
)

server <- function(input, output, session) {
  observeEvent(input$recalculate, {
    output$numbers <- renderText({
      rnorm(input$n)
    })
  })
}

shinyApp(ui, server)
```



Solution on next page



Don't overuse `observeEvent()` - this is a very common anti-pattern. Do not place renders inside observers.

```
library(shiny)

ui <- fluidPage(
  numericInput("n", "Number of observations", 5),
  actionButton("recalculate", "Recalculate"),
  verbatimTextOutput("numbers")
)

server <- function(input, output, session) {
  output$numbers <- renderText({
    input$recalculate
    isolate(rnorm(input$n))
  })
}

shinyApp(ui, server)
```



Update Functions

- Every input function has a corresponding update function

<code>textInput()</code>	<code>updateTextInput()</code>
<code>sliderInput()</code>	<code>updateSliderInput()</code>
<code>radioButtons()</code>	<code>updateRadioButtons()</code>

- Must define server as

```
function(input, output, session)
```

This is the app we used to learn about `uiOutput()`. Every time the minimum changes, the maximum resets back 100, which can be annoying.

```
ui <- fluidPage(  
  sliderInput(  
    "min", "Minimum", min = 0, max = 99, value = 0  
  ),  
  uiOutput("max_input")  
)  
  
server <- function(input, output) {  
  output$max_input <- renderUI({  
    sliderInput(  
      "max", "Maximum",  
      min = input$min,  
      max = 100, value = 100  
    )  
  })  
}  
  
shinyApp(ui, server)
```

Fix this behaviour so that the maximum doesn't change when the minimum does.

Try to solve it first without update functions and then with them.



Solution on next page

Without update function

```
ui <- fluidPage(
  sliderInput(
    "min", "Minimum", min = 0, max = 99, value = 0
  ),
  uiOutput("max_input")
)

server <- function(input, output) {
  output$max_input <- renderUI({
    if (is.null(input$max)) {
      value <- 100
    } else {
      value <- input$max
    }
    sliderInput(
      "max", "Maximum",
      min = input$min,
      max = 100, value = value
    )
  })
}

shinyApp(ui, server)
```



With update function

```
ui <- fluidPage(
  sliderInput(
    "min", "Minimum", min = 0, max = 99, value = 0
  ),
  sliderInput(
    "max", "Maximum", min = 0, max = 100, value = 100
  )
)

server <- function(input, output, session) {
  observeEvent(input$min, {
    updateSliderInput(session, "max", min = input$min)
  })
}

shinyApp(ui, server)
```

- Whenever possible, define inputs in UI and only update instead of rendering whole inputs in server
- Clearer to read, you know what all the inputs are when looking at the UI
- Better performance, less data sending between server and client, less work by browser
- Only parameter you want is changed, so previous value can be kept easily

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)

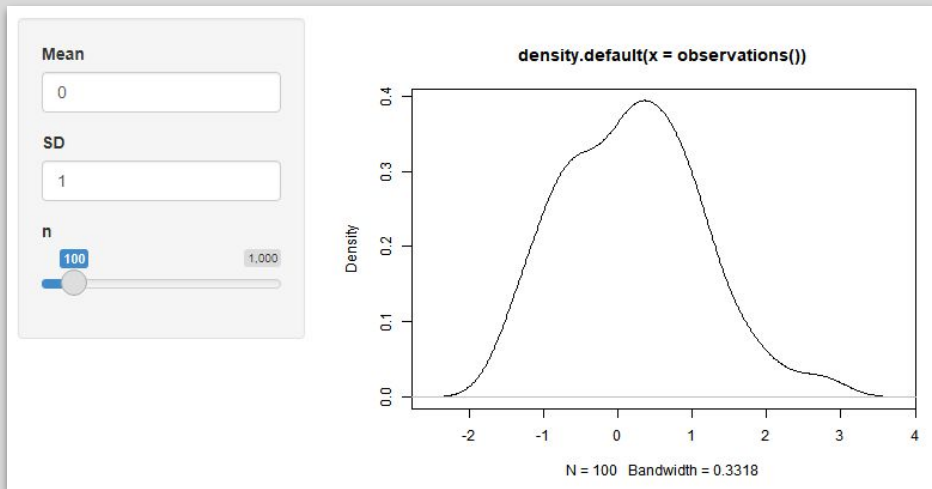
renderPlot({
 plot(seq(input$num))
})
```,
```

Final exercise

Build a simple tool that helps stats students see what a Normal distribution looks like. We'll build it in steps.

Step 1:

- Three inputs: mean, SD, n
- Plot the density distribution `plot(density(rnorm(...)))`
- Use a sidebar layout
- Notice the slider has no tick marks



Solution on next page



```
fluidPage(
  sidebarLayout(
    sidebarPanel(
      numericInput("mean", "Mean", 0),
      numericInput("sd", "SD", 1),
      sliderInput("nobs", "n", min = 10, max = 1000,
        value = 100, ticks = FALSE)
    ),
    mainPanel(
      plotOutput("plot")
    )
  )
)
```

ui.R

```
function(input, output, session) {
  observations <- reactive({
    rnorm(input$nobs, input$mean, input$sd)
  })

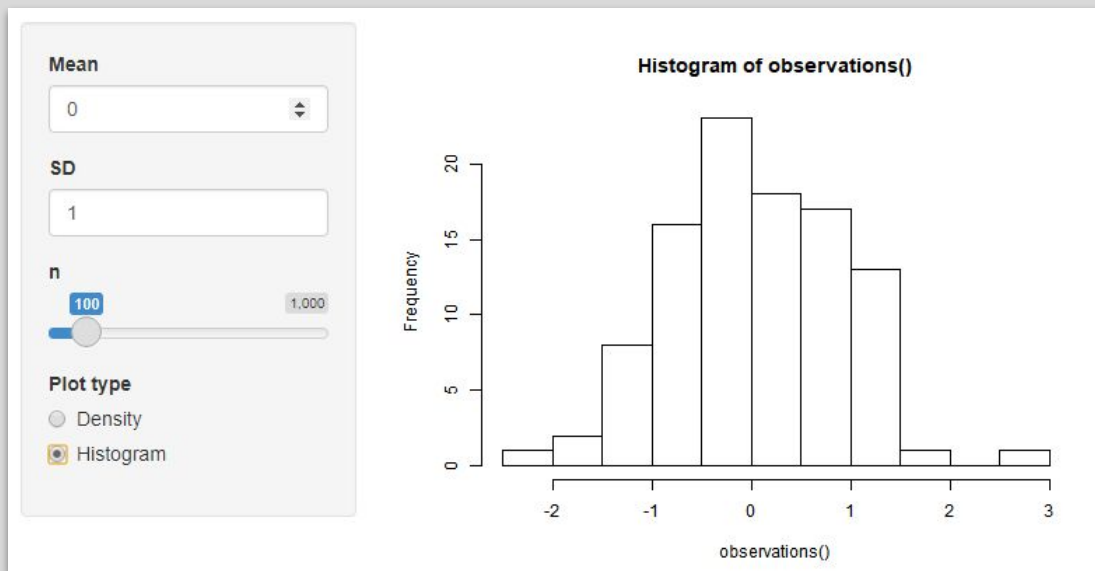
  plot_data <- reactive({
    density(observations())
  })

  output$plot <- renderPlot({
    plot(plot_data())
  })
}
```

server.R

Step 2:

- Add an option to plot a histogram instead of density using radio buttons
`plot(hist(rnorm(...)))`



Solution on next page




```
fluidPage(
  sidebarLayout(
    sidebarPanel(
      numericInput("mean", "Mean", 0),
      numericInput("sd", "SD", 1),
      sliderInput("nobs", "n", min = 10, max = 1000,
        value = 100, ticks = FALSE),
      radioButtons("type", "Plot type",
        c("Density", "Histogram"))
    ),
    mainPanel(
      plotOutput("plot")
    )
  )
)
```

ui.R

```
function(input, output, session) {
  observations <- reactive({
    rnorm(input$nobs, input$mean, input$sd)
  })

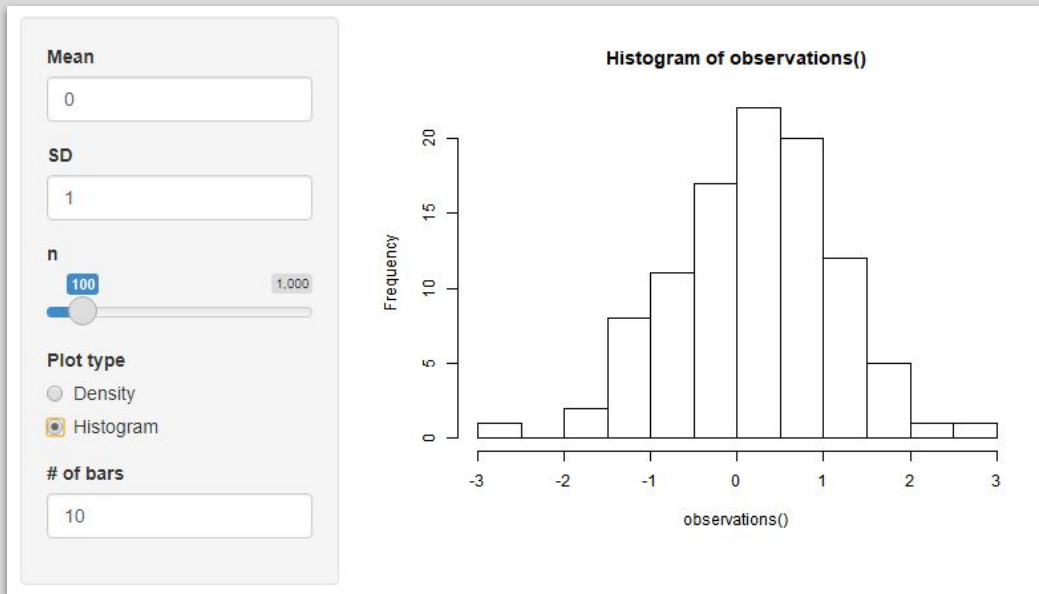
  plot_data <- reactive({
    if (input$type == "Density") {
      density(observations())
    } else if (input$type == "Histogram") {
      hist(observations())
    }
  })

  output$plot <- renderPlot({
    plot(plot_data())
  })
}
```

server.R

Step 3:

- Add a “# of bars” numeric input that only applies to histograms
- Use it with the `breaks` parameter of `hist()` to control the number of bars



Solution on next page



ui.R

```
fluidPage(
  sidebarLayout(
    sidebarPanel(
      numericInput("mean", "Mean", 0),
      numericInput("sd", "SD", 1),
      sliderInput("nobs", "n", min = 10, max = 1000,
        value = 100, ticks = FALSE),
      radioButtons("type", "Plot type",
        c("Density", "Histogram")),
      conditionalPanel(
        "input.type == 'Histogram'",
        numericInput("nbars", "# of bars", 10)
      )
    ),
    mainPanel(
      plotOutput("plot")
    )
  )
)
```

server.R

```
function(input, output, session) {
  observations <- reactive({
    rnorm(input$nobs, input$mean, input$sd)
  })

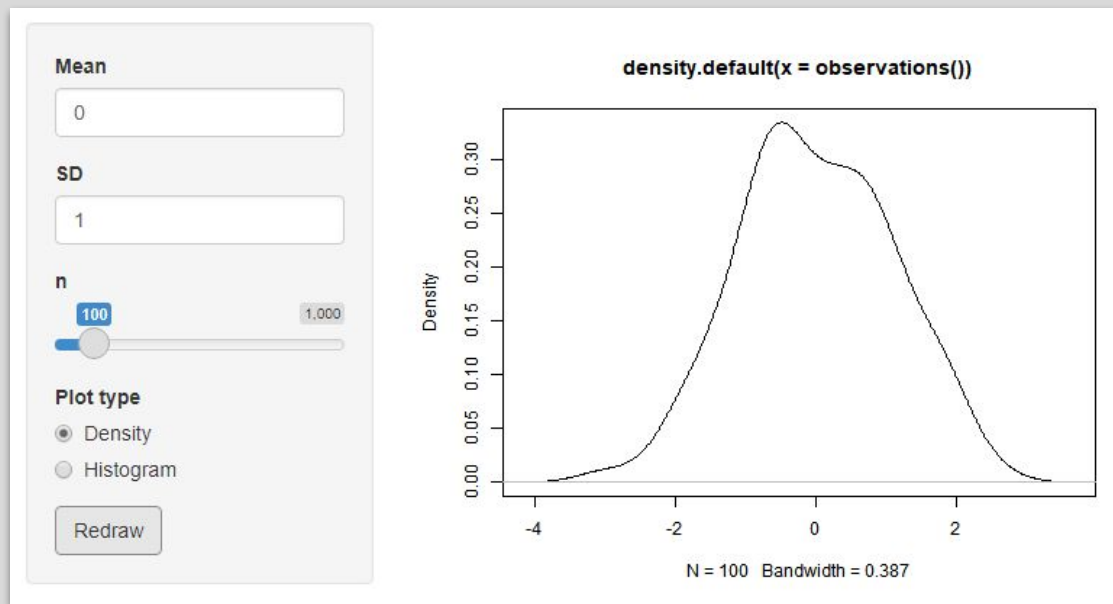
  plot_data <- reactive({
    if (input$type == "Density") {
      density(observations())
    } else if (input$type == "Histogram") {
      hist(observations(), breaks = input$nbars)
    }
  })

  output$plot <- renderPlot({
    plot(plot_data())
  })
}
```



Step 4:

- Add a “Redraw” button that redraws the plot with new data



Solution on next page



```
fluidPage(
  sidebarLayout(
    sidebarPanel(
      numericInput("mean", "Mean", 0),
      numericInput("sd", "SD", 1),
      sliderInput("nobs", "n", min = 10, max = 1000,
        value = 100, ticks = FALSE),
      radioButtons("type", "Plot type",
        c("Density", "Histogram")),
      conditionalPanel(
        "input.type == 'Histogram'",
        numericInput("nbars", "# of bars", 10)
      ),
      actionButton("redraw", "Redraw")
    ),
    mainPanel(
      plotOutput("plot")
    )
  )
)
```

ui.R

```
function(input, output, session) {
  observations <- reactive({
    input$redraw
    rnorm(input$nobs, input$mean, input$sd)
  })

  plot_data <- reactive({
    if (input$type == "Density") {
      density(observations())
    } else if (input$type == "Histogram") {
      hist(observations(), breaks = input$nbars)
    }
  })

  output$plot <- renderPlot({
    plot(plot_data())
  })
}
```

server.R

Step 5:

- Only redraw the plot when the button is pressed
- When the app starts, do not show a plot until the button is pressed



Solution on next page



ui.R

```
fluidPage(
  sidebarLayout(
    sidebarPanel(
      numericInput("mean", "Mean", 0),
      numericInput("sd", "SD", 1),
      sliderInput("nobs", "n", min = 10, max = 1000,
        value = 100, ticks = FALSE),
      radioButtons("type", "Plot type",
        c("Density", "Histogram")),
      conditionalPanel(
        "input.type == 'Histogram'",
        numericInput("nbars", "# of bars", 10)
      ),
      actionButton("redraw", "Redraw")
    ),
    mainPanel(
      plotOutput("plot")
    )
  )
)
```

server.R

```
function(input, output, session) {
  observations <- reactive({
    req(input$redraw)
    isolate({
      rnorm(input$nobs, input$mean, input$sd)
    })
  })

  plot_data <- reactive({
    if (input$type == "Density") {
      density(observations())
    } else if (input$type == "Histogram") {
      hist(observations(), breaks = isolate(input$nbars))
    }
  })

  output$plot <- renderPlot({
    plot(plot_data())
  })
}
```



Feedback please!*



<https://forms.gle/HYrny74Ew1GYCrrU7>

*Constructive criticism is appreciated - unleash your inner demon 🐈