Install instructions

```
install.packages(c(‘shiny’,’bslib’,’reactlog’))
```

Building R-Shiny Applications

FAS6932 – Special Topics

Summer C 2025

Welcome!

Course/workshop schedule

- Meeting times: 9a - ~3p 5/19-5/21, 5/27-5/28
- Meeting location: ACF: 5/19–5/21, Millhopper: 5/27–5/28
- Course progression:
  - Day 1: Basics
  - Day 2: Design
  - Day 3: Reactivity
  - Day 4: Advanced design
  - Day 5: Hosting

- **SYLLABUS!**

Icebreaker

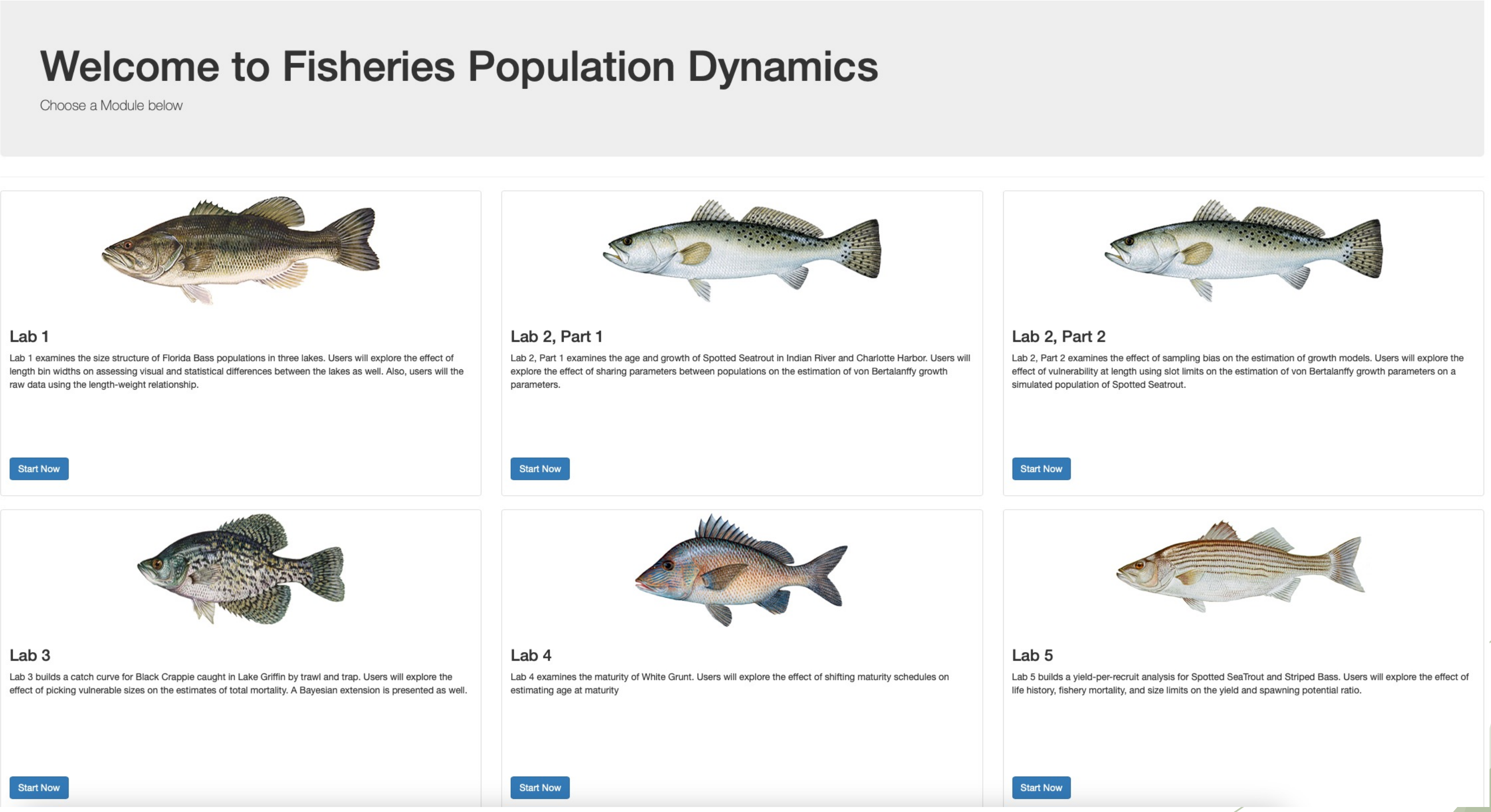
- Introduce yourself!

- **Students**: Name, department, degree
- **Workshoppers**: Name, employer, job

- **All:**
  - Who is your intended audience?
    - You? Colleagues? Scientific journal? Managers? Public?
  - What R-Shiny experience (or PowerBI, ArcOnline dashboard) do you have?

About me

https://zsiders.shinyapps.io/Welcome_Module/



Welcome to Fisheries Population Dynamics

Choose a Module below

Lab 1

Lab 1 examines the size structure of Florida Bass populations in three lakes. Users will explore the effect of length bin widths on assessing visual and statistical differences between the lakes as well. Also, users will the raw data using the length-weight relationship.

Lab 2, Part 1

Lab 2, Part 1 examines the age and growth of Spotted Seatrout in Indian River and Charlotte Harbor. Users will explore the effect of sharing parameters between populations on the estimation of von Bertalanffy growth parameters.

Lab 2, Part 2

Lab 2, Part 2 examines the effect of sampling bias on the estimation of growth models. Users will explore the effect of vulnerability at length using slot limits on the estimation of von Bertalanffy growth parameters on a simulated population of Spotted Seatrout.

Lab 3

Lab 3 builds a catch curve for Black Crappie caught in Lake Griffin by trawl and trap. Users will explore the effect of picking vulnerable sizes on the estimates of total mortality. A Bayesian extension is presented as well.

Lab 4

Lab 4 examines the maturity of White Grunt. Users will explore the effect of shifting maturity schedules on estimating age at maturity

Lab 5

Lab 5 builds a yield-per-recruit analysis for Spotted SeaTrout and Striped Bass. Users will explore the effect of life history, fishery mortality, and size limits on the yield and spawning potential ratio.

Day 1 Schedule:

- Morning Session
  - Course introduction
  - Basics of Shiny applications
  - Resources
- Lunch
- Afternoon Session
  - Practice building a basic application
  - Small group breakout

Course website

https://github.com/zsiders/BuildingRShinyCourse

Resources

- https://shiny.posit.co/r/getstarted/shiny-basics/lesson1/
- https://mastering-shiny.org/index.html
- https://github.com/nanxstats/awesome-shiny-extensions

What’s in an app?

- Two components:
- User Interface (UI)
  - Everything you want a user to see and interact with
  - Fairly rigid and can be challenging to be reactive (Chp. 10 in Mastering Shiny)
  - Prone to faults
- Server
  - All your reactions to input
    - Includes filtering, calculations, entire models if you want
  - Any visualizations or tabulations that you want to render

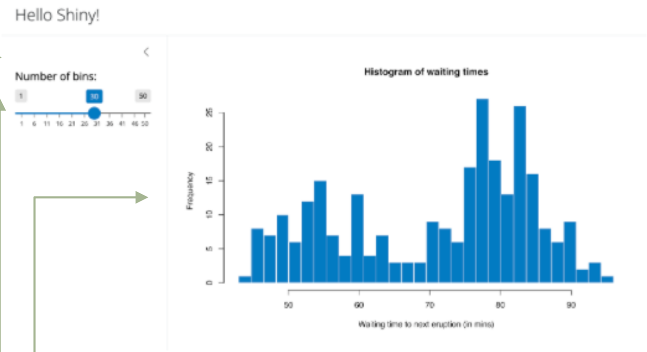
What’s in an app?

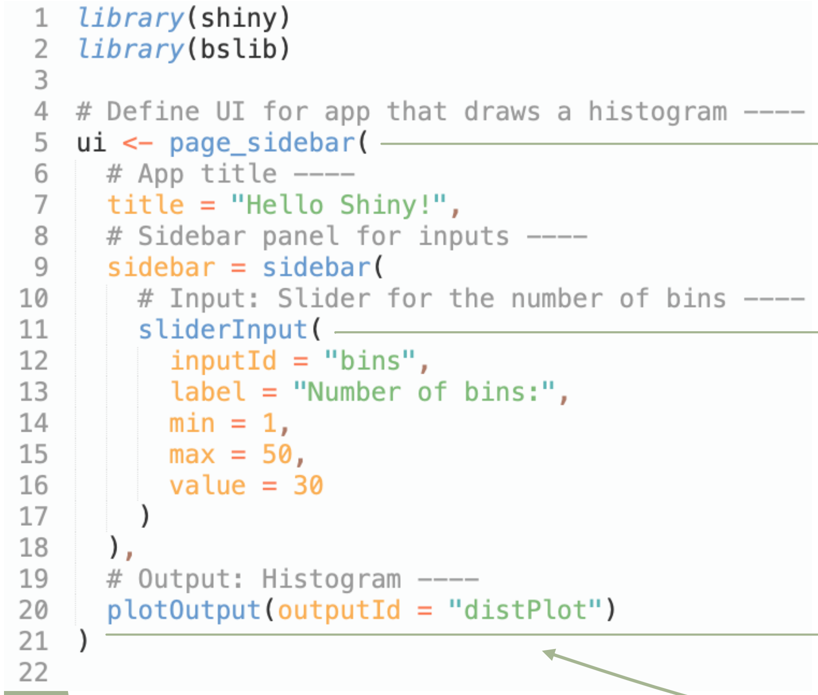
- Two components:
- User Interface (UI)
  - Everything you want a user to see and interact with
  - Fairly rigid and can be challenging to be reactive (Chp. 10 in Mastering Shiny)
  - Prone to faults
- Server
  - All your reactions to input
    - Includes filtering, calculations, entire models if you want
  - Any visualizations or tabulations that you want to render

RUN:

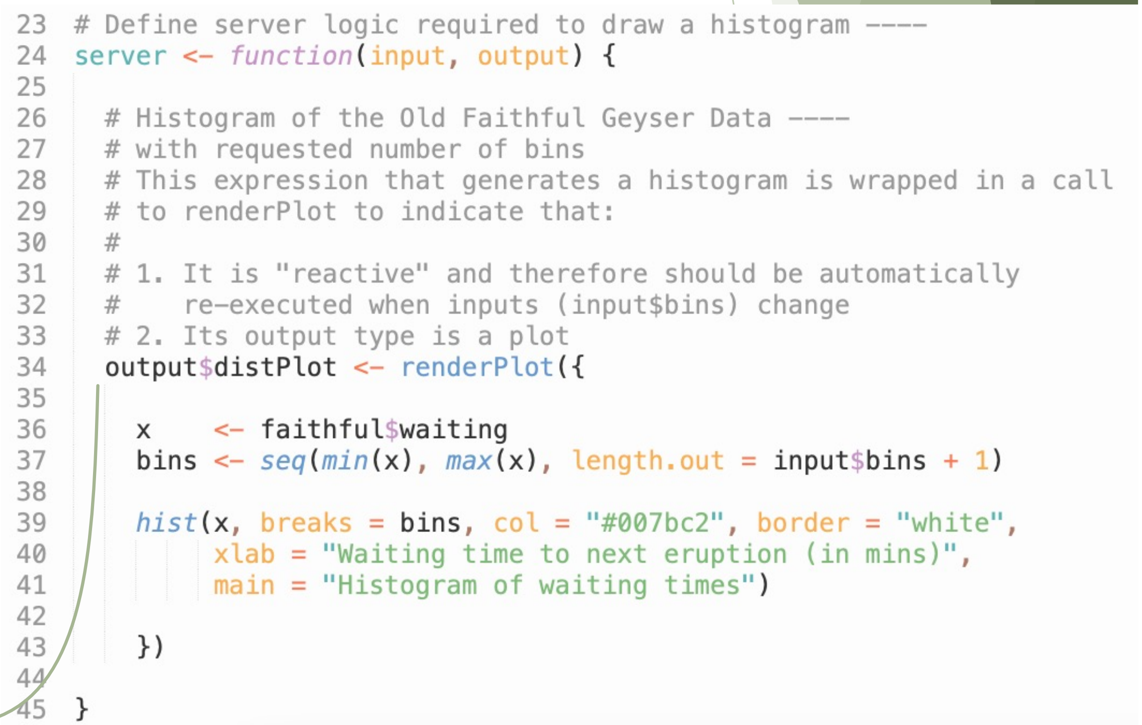
```
library(shiny)
runExample("01_hello")
```

What’s in an app?





```r
library(shiny)
library(bslib)

# Define UI for app that draws a histogram ----
ui <- page_sidebar(
  # App title ----
  title = "Hello Shiny!",
  # Sidebar panel for inputs ----
  sidebar = sidebar(
    # Input: Slider for the number of bins ----
    sliderInput(
      inputId = "bins",
      label = "Number of bins:",
      min = 1,
      max = 50,
      value = 30
    )
  ),
  # Output: Histogram ----
  plotOutput(outputId = "distPlot")
)

```



```r
# Define server logic required to draw a histogram ----
server <- function(input, output) {

  # Histogram of the Old Faithful Geyser Data ----
  # with requested number of bins
  # This expression that generates a histogram is wrapped in a call
  # to renderPlot to indicate that:
  #
  # 1. It is "reactive" and therefore should be automatically
  #    re-executed when inputs (input$bins) change
  # 2. Its output type is a plot
  output$distPlot <- renderPlot({

    x    <- faithful$waiting
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    hist(x, breaks = bins, col = "#007bc2", border = "white",
         xlab = "Waiting time to next eruption (in mins)",
         main = "Histogram of waiting times")

    })

}
```

User interface

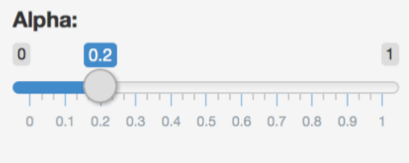
- Three components:
  - **Page layout** (https://shiny.posit.co/r/layouts/)
    - Some extensions for more design options
  - **Interactive widgets** (https://shiny.posit.co/r/components/)
    - Also where most of the extensions have been made
  - **Content**
    - Visualizations
    - Tabulations

- In R, this is a call-specific list object
- It also generates a secondary list object called *inputs*

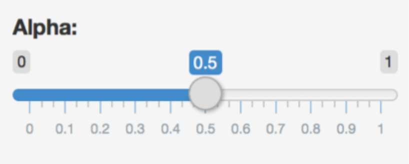
UI to input list

```
# Set alpha level
sliderInput(inputId = "alpha",
            label = "Alpha:",
            min = 0, max = 1,
            value = 0.5)
```

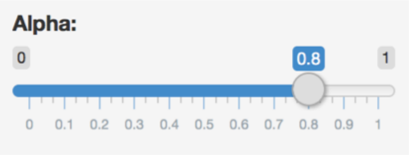
input$alpha



input$alpha = 0.2

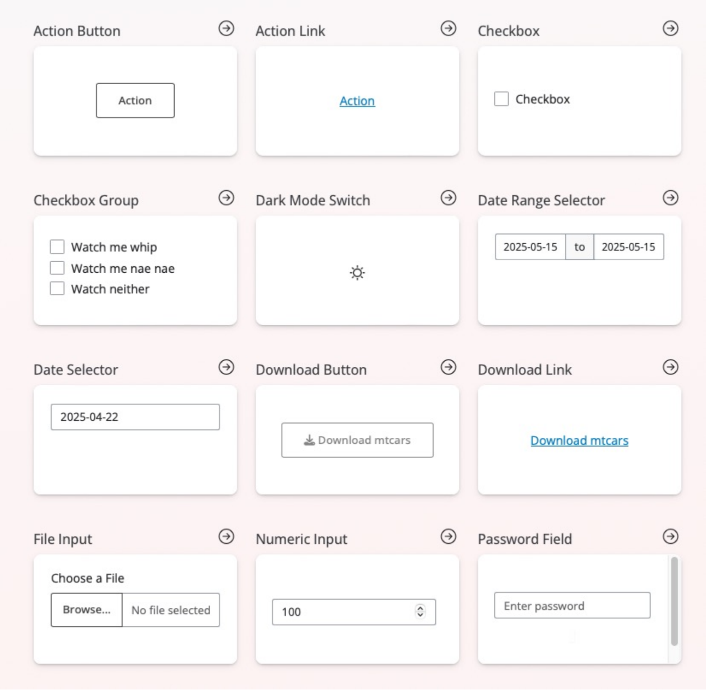


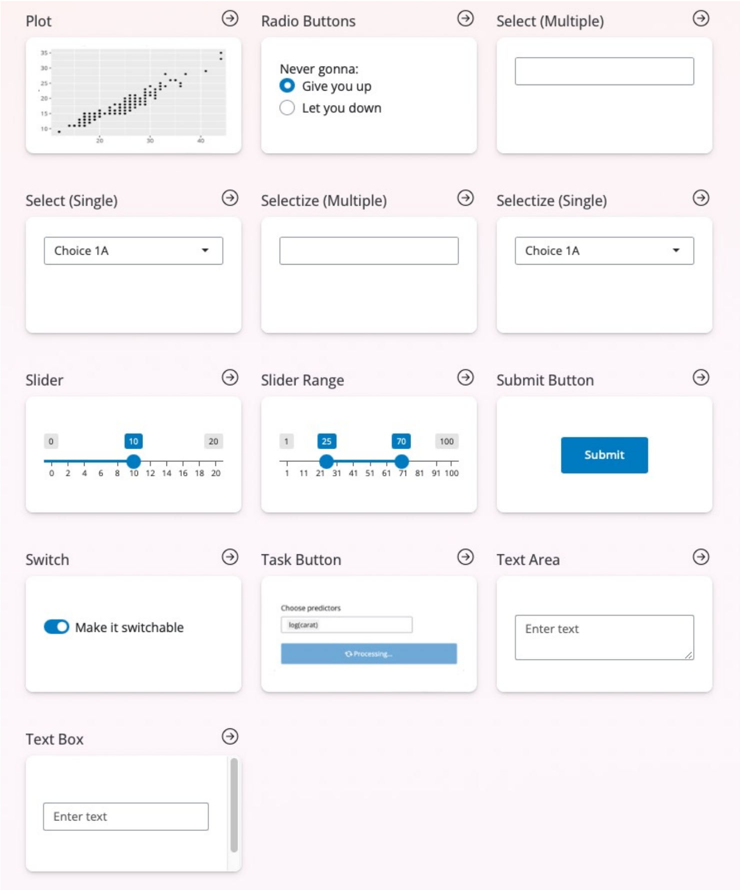
input$alpha = 0.5



input$alpha = 0.8

Widgets!





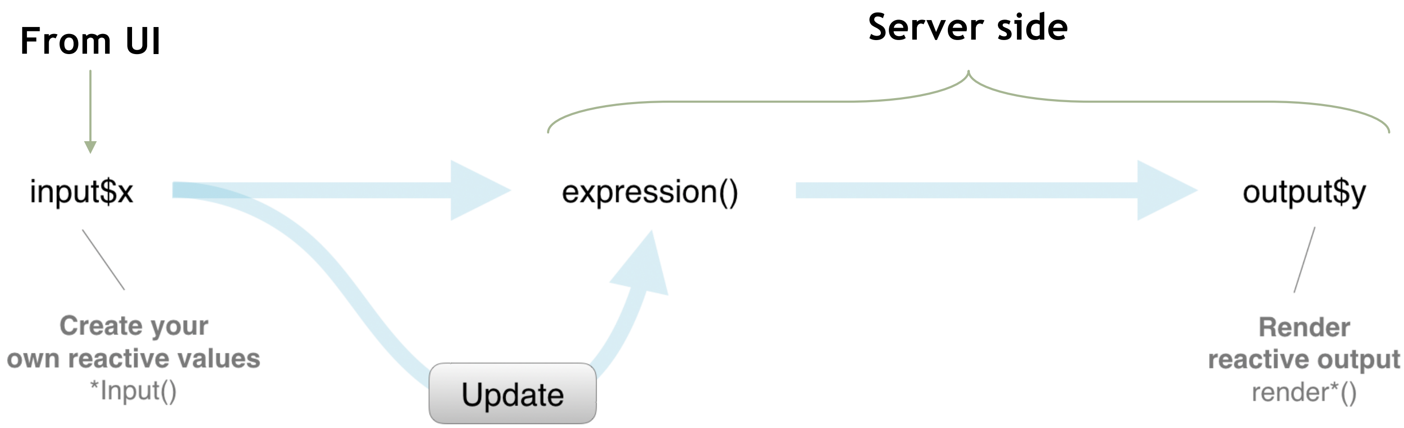
https://github.com/nanxstats/awesome-shiny-extensions

Server

- This is a function!
  - Takes arguments: input, output, and session
- **Input** is a list object that is generated from the UI widgets (and user's cursor)
- **Output** is a list object that is generated from the server
  - anything that you want rendered in the UI (like plots!)
- **Session** is passed to the server function from the *shinyApp* function
  - Basically, makes an instance of the Shiny application

- Server-side expressions **react** to UI inputs

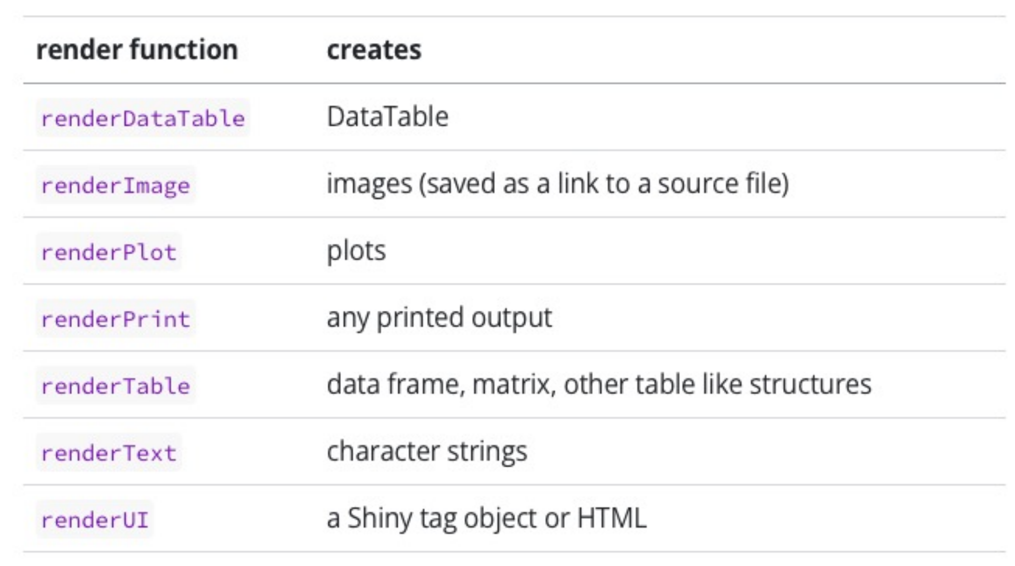
Basic reactivity



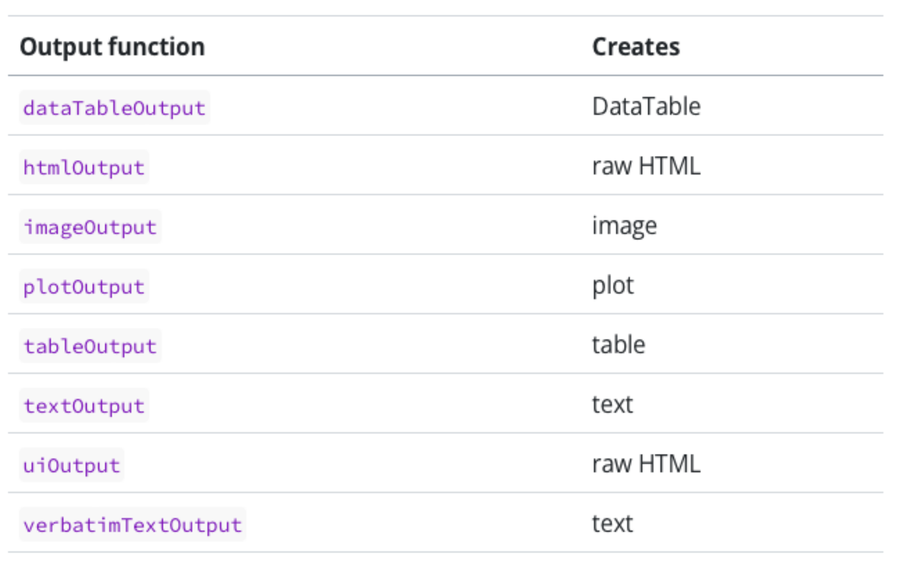
The simplest reactivity uses built-in render functions

To make these functions reactive, we merely need to call an object from the the *input* list

Linking UI to Server output

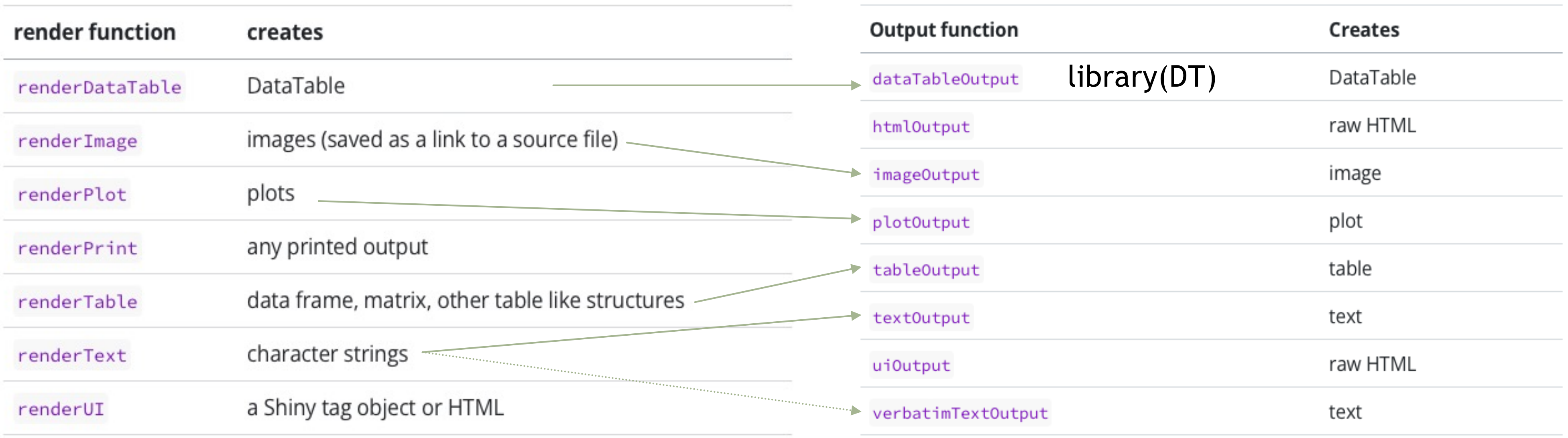


render function	creates
`renderDataTable`	DataTable
`renderImage`	images (saved as a link to a source file)
`renderPlot`	plots
`renderPrint`	any printed output
`renderTable`	data frame, matrix, other table like structures
`renderText`	character strings
`renderUI`	a Shiny tag object or HTML



Output function	Creates
`dataTableOutput`	DataTable
`htmlOutput`	raw HTML
`imageOutput`	image
`plotOutput`	plot
`tableOutput`	table
`textOutput`	text
`uiOutput`	raw HTML
`verbatimTextOutput`	text

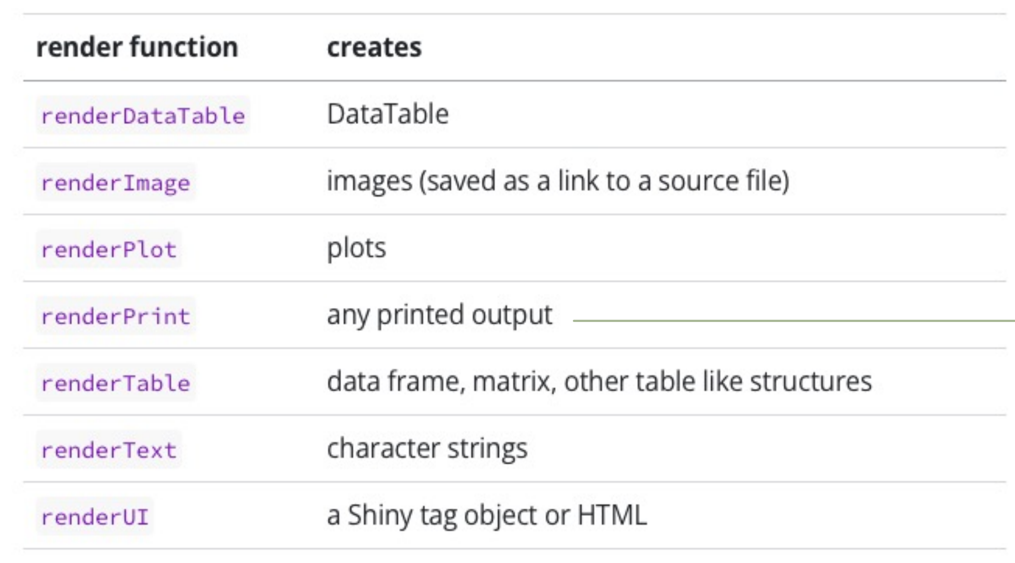
Linking UI to Server output



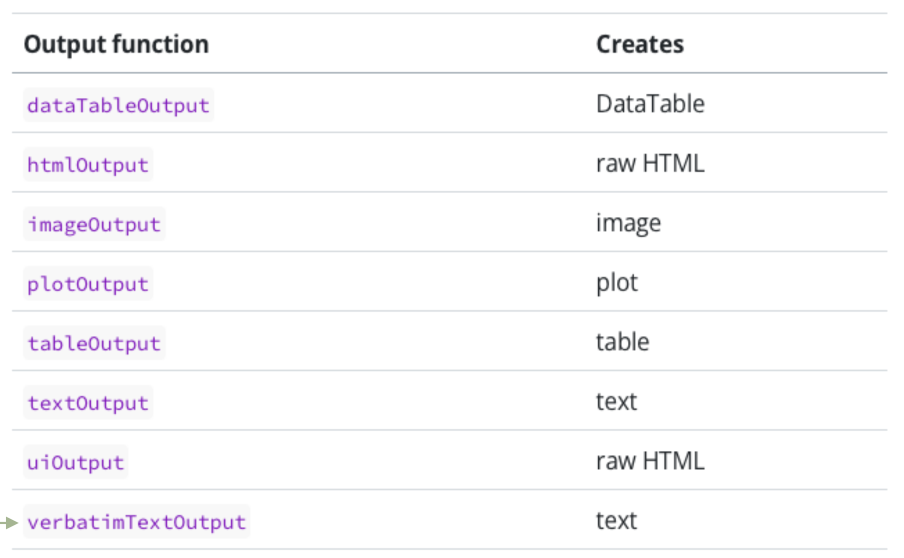
render function	creates
`renderDataTable`	DataTable
`renderImage`	images (saved as a link to a source file)
`renderPlot`	plots
`renderPrint`	any printed output
`renderTable`	data frame, matrix, other table like structures
`renderText`	character strings
`renderUI`	a Shiny tag object or HTML

Output function		Creates
`dataTableOutput`	library(DT)	DataTable
`htmlOutput`		raw HTML
`imageOutput`		image
`plotOutput`		plot
`tableOutput`		table
`textOutput`		text
`uiOutput`		raw HTML
`verbatimTextOutput`		text

Linking UI to Server output

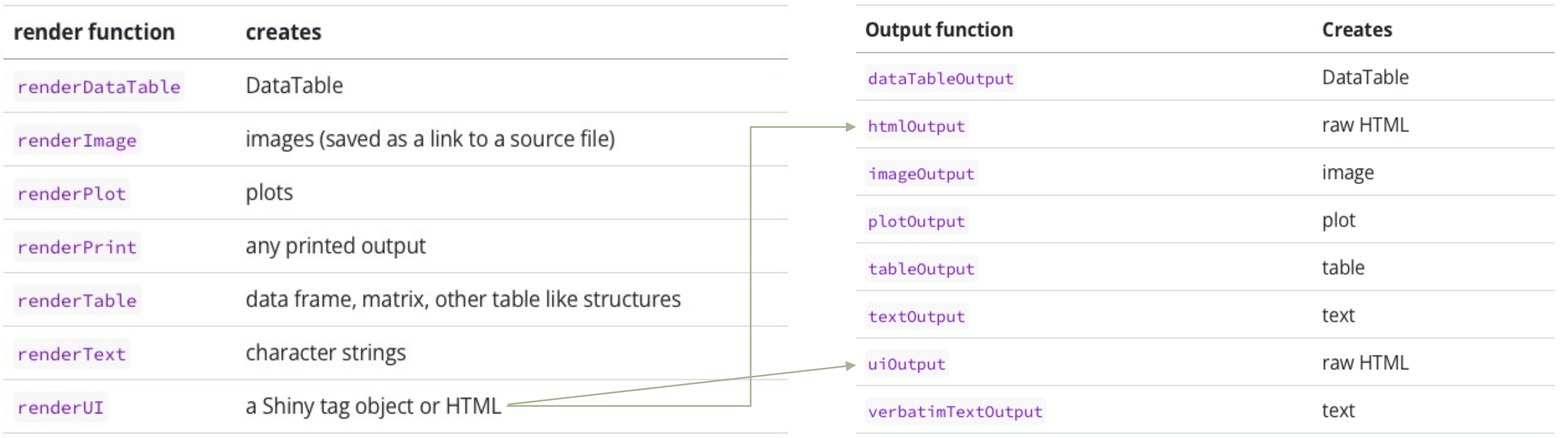


render function	creates
`renderDataTable`	DataTable
`renderImage`	images (saved as a link to a source file)
`renderPlot`	plots
`renderPrint`	any printed output
`renderTable`	data frame, matrix, other table like structures
`renderText`	character strings
`renderUI`	a Shiny tag object or HTML



Output function	Creates
`dataTableOutput`	DataTable
`htmlOutput`	raw HTML
`imageOutput`	image
`plotOutput`	plot
`tableOutput`	table
`textOutput`	text
`uiOutput`	raw HTML
`verbatimTextOutput`	text

Linking UI to Server output



render function	creates
`renderDataTable`	DataTable
`renderImage`	images (saved as a link to a source file)
`renderPlot`	plots
`renderPrint`	any printed output
`renderTable`	data frame, matrix, other table like structures
`renderText`	character strings
`renderUI`	a Shiny tag object or HTML

Output function	Creates
`dataTableOutput`	DataTable
`htmlOutput`	raw HTML
`imageOutput`	image
`plotOutput`	plot
`tableOutput`	table
`textOutput`	text
`uiOutput`	raw HTML
`verbatimTextOutput`	text

Let’s see an example!

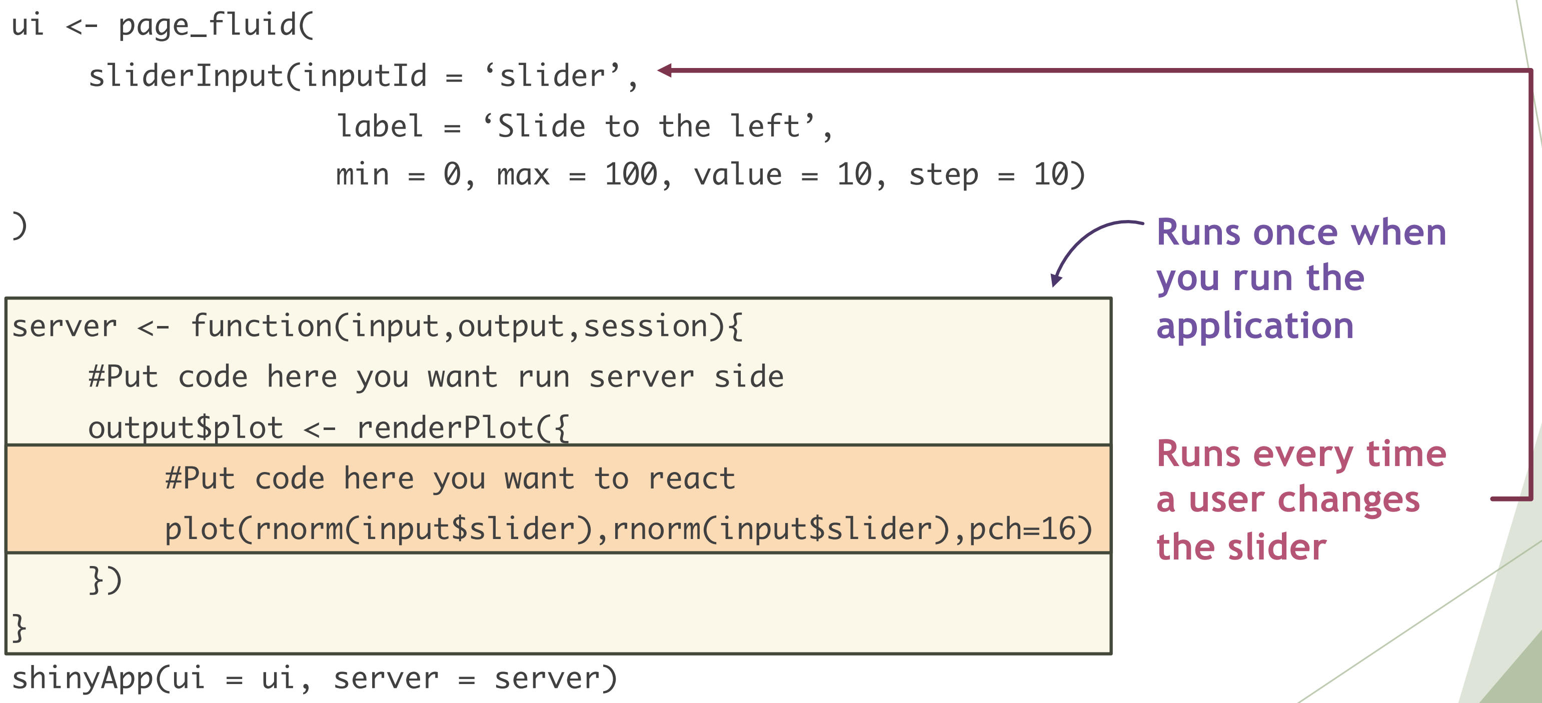
Run render2output.R

10 minute break

Let’s dissect an app!

Run lizards_of_the_world.R

Structure of app code



```
# Put code here that does any data preparation or any functions you want to use
(can also be in a separate file)
ui <- page_fluid(
    sliderInput(inputId = ‘slider’,
                label = ‘Slide to the left’,
                min = 0, max = 100, value = 10, step = 10)
)
server <- function(input,output,session){
    #Put code here you want run server side
    output$plot <- renderPlot({
        #Put code here you want to react
        plot(rnorm(input$slider),rnorm(input$slider),pch=16)
    })
}
shinyApp(ui = ui, server = server)
```

Runs once when you run the application

Runs every time a user changes the slider

Structure of app code

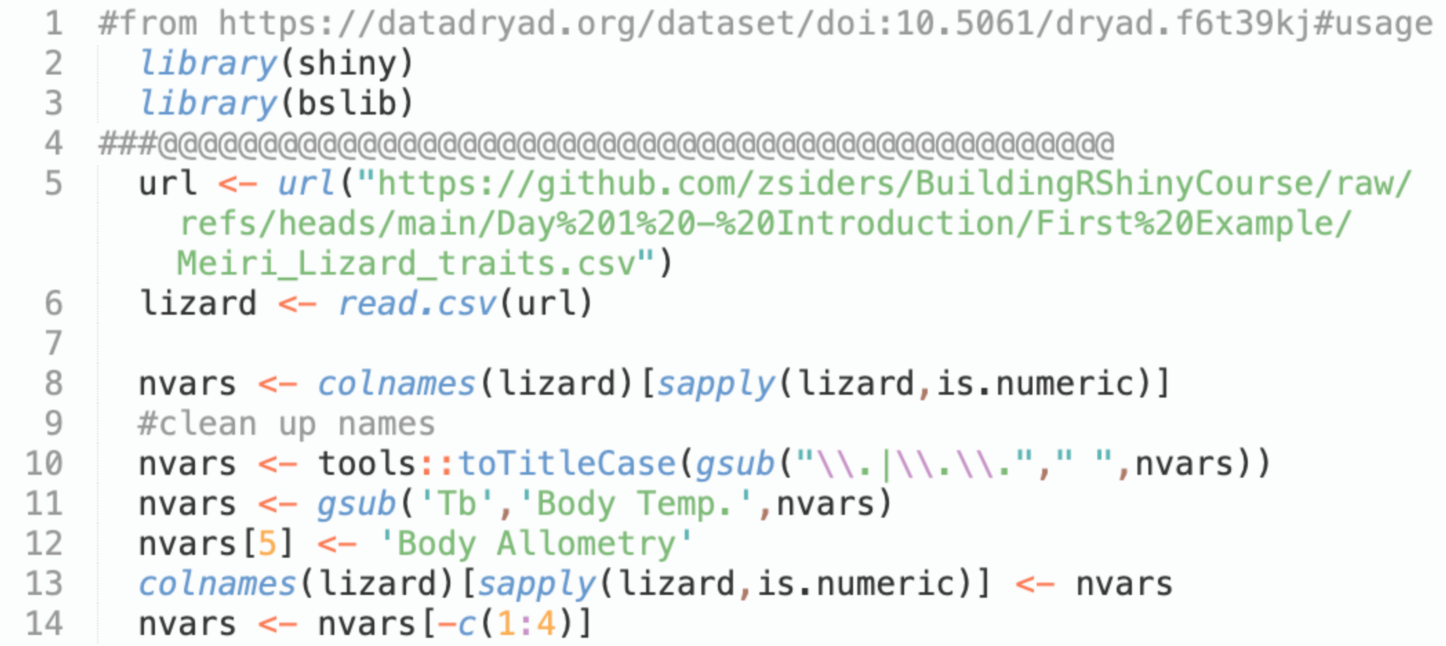
```
# Put code here that does any data preparation or any functions you want to use
(can also be in a separate file)
ui <- page_fluid(
    sliderInput(inputId = ‘slider’,
                label = ‘Slide to the left’,
                min = 0, max = 100, value = 10, step = 10)
)
server <- function(input,output,session){
    #Put code here you want run server side
    output$plot <- renderPlot({
        #Put code here you want to react
        plot(rnorm(input$slider),rnorm(input$slider),pch=16)
    })
}
shinyApp(ui = ui, server = server)
```

Runs once when you run the application

Runs every time a user changes the slider

The server is a watcher once run. It is the checkpoint for changes in the UI (user input) and sends along the notice to reactive components

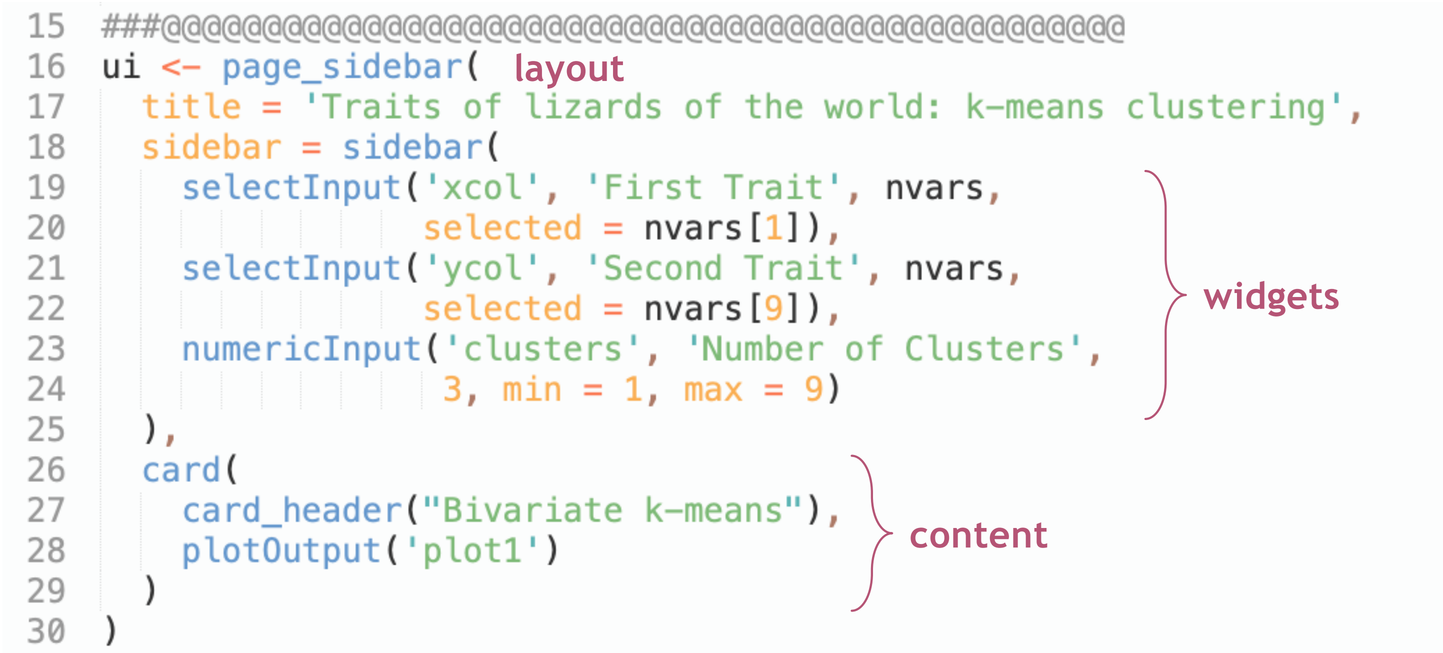
Section 1 – lizards_of_the_world.R



```r
#from https://datadryad.org/dataset/doi:10.5061/dryad.f6t39kj#usage
  library(shiny)
  library(bslib)
###@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  url <- url("https://github.com/zsiders/BuildingRShinyCourse/raw/refs/heads/main/Day%201%20-%20Introduction/First%20Example/Meiri_Lizard_traits.csv")
  lizard <- read.csv(url)

  nvars <- colnames(lizard)[sapply(lizard,is.numeric)]
  #clean up names
  nvars <- tools::toTitleCase(gsub("\\.|\\.\\."," ",nvars))
  nvars <- gsub('Tb','Body Temp.',nvars)
  nvars[5] <- 'Body Allometry'
  colnames(lizard)[sapply(lizard,is.numeric)] <- nvars
  nvars <- nvars[-c(1:4)]
```

This is making objects that are in the GlobalEnvironment that can be access by the app anywhere in the code

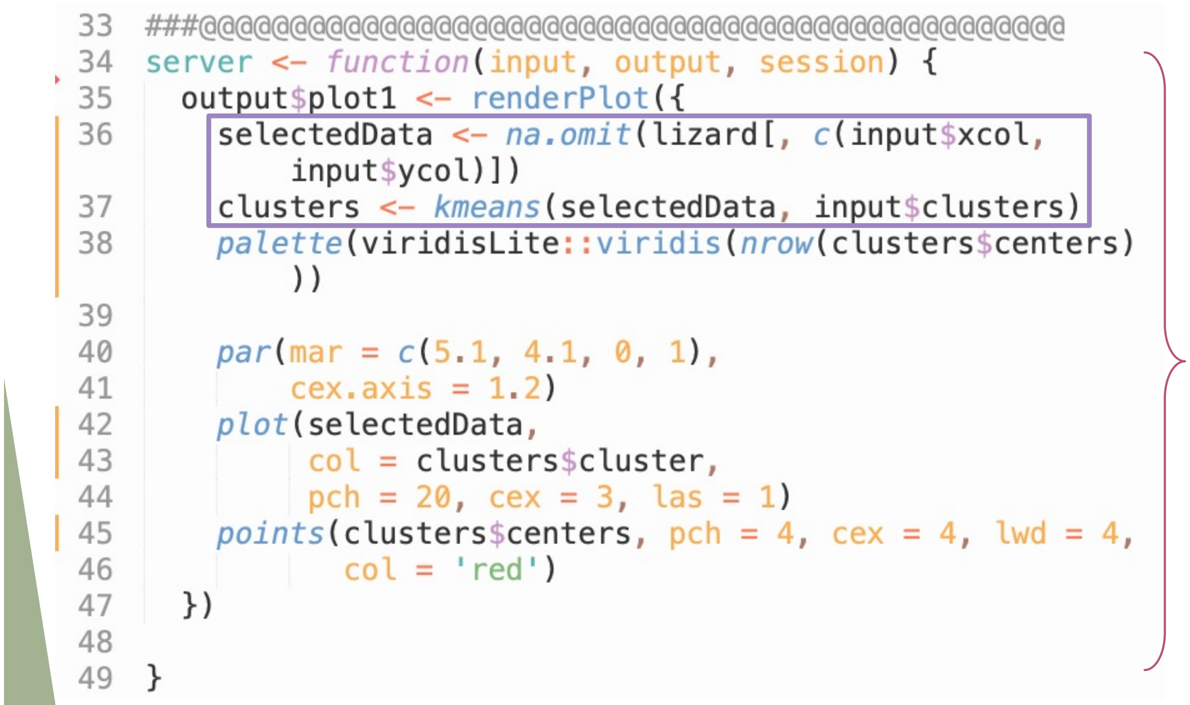
Section 2



```r
###@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
ui <- page_sidebar(
  title = 'Traits of lizards of the world: k-means clustering',
  sidebar = sidebar(
    selectInput('xcol', 'First Trait', nvars,
                selected = nvars[1]),
    selectInput('ycol', 'Second Trait', nvars,
                selected = nvars[9]),
    numericInput('clusters', 'Number of Clusters',
                 3, min = 1, max = 9)
  ),
  card(
    card_header("Bivariate k-means"),
    plotOutput('plot1')
  )
)
```

This setups the UI. We could subdivide the layout blocks (sidebar, content) further if we wanted

Section 3

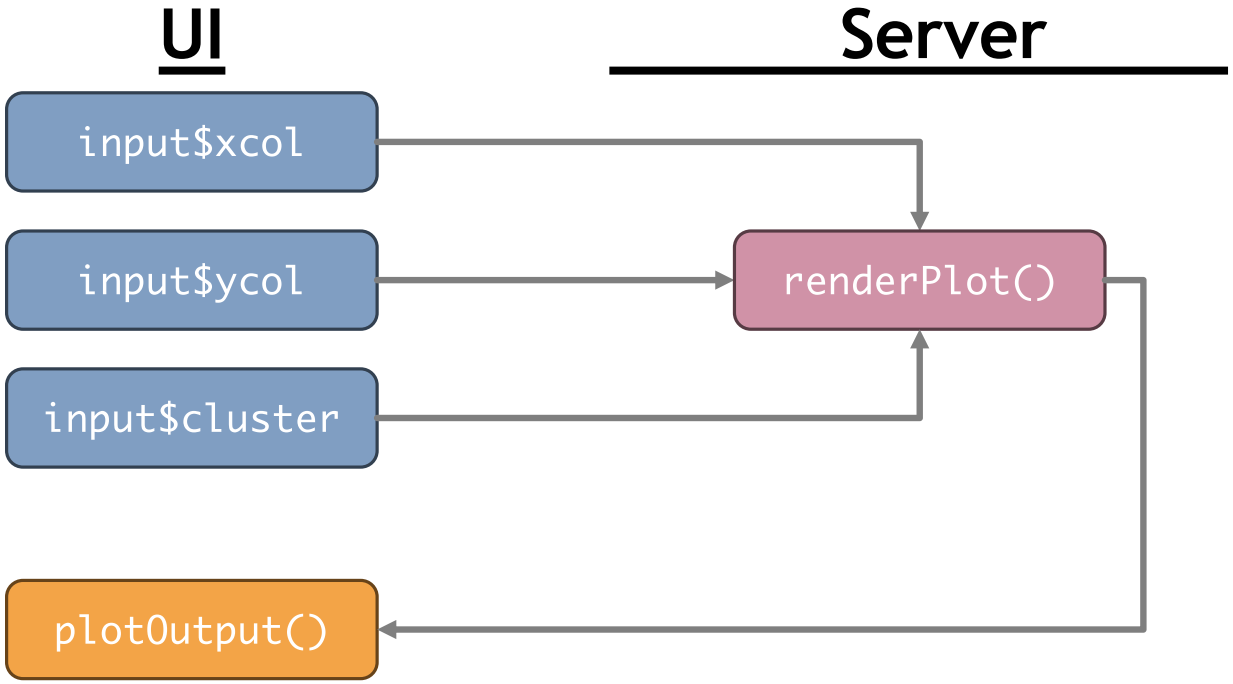


```
###@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
server <- function(input, output, session) {
  output$plot1 <- renderPlot({
    selectedData <- na.omit(lizard[, c(input$xcol,
        input$ycol)])
    clusters <- kmeans(selectedData, input$clusters)
    palette(viridisLite::viridis(nrow(clusters$centers)
        ))

    par(mar = c(5.1, 4.1, 0, 1),
        cex.axis = 1.2)
    plot(selectedData,
         col = clusters$cluster,
         pch = 20, cex = 3, las = 1)
    points(clusters$centers, pch = 4, cex = 4, lwd = 4,
           col = 'red')
  })

}
```

this is a reactive <u>endpoint!</u>

Notice in this one we react to multiple input values

Breaking down that reactivity a bit more…



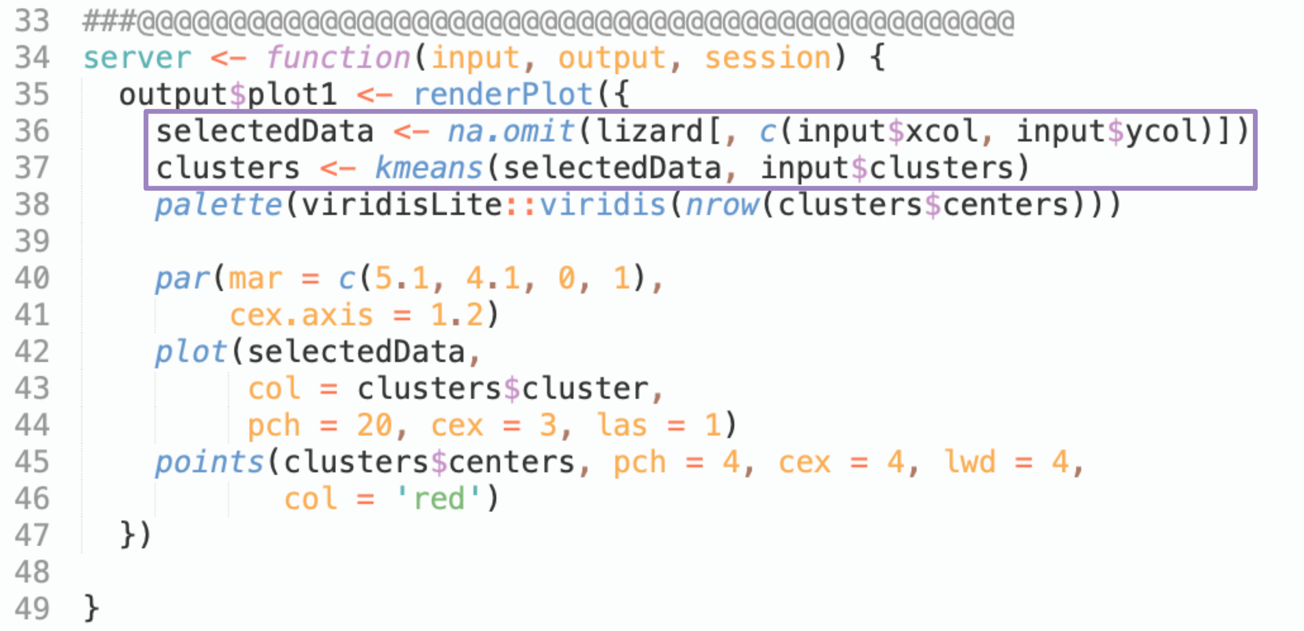
Let’s use reactlog to track our reactivity

Run lizards_betterreact.R

https://rstudio.github.io/reactlog/articles/reactlog.html

Reactivity challenges

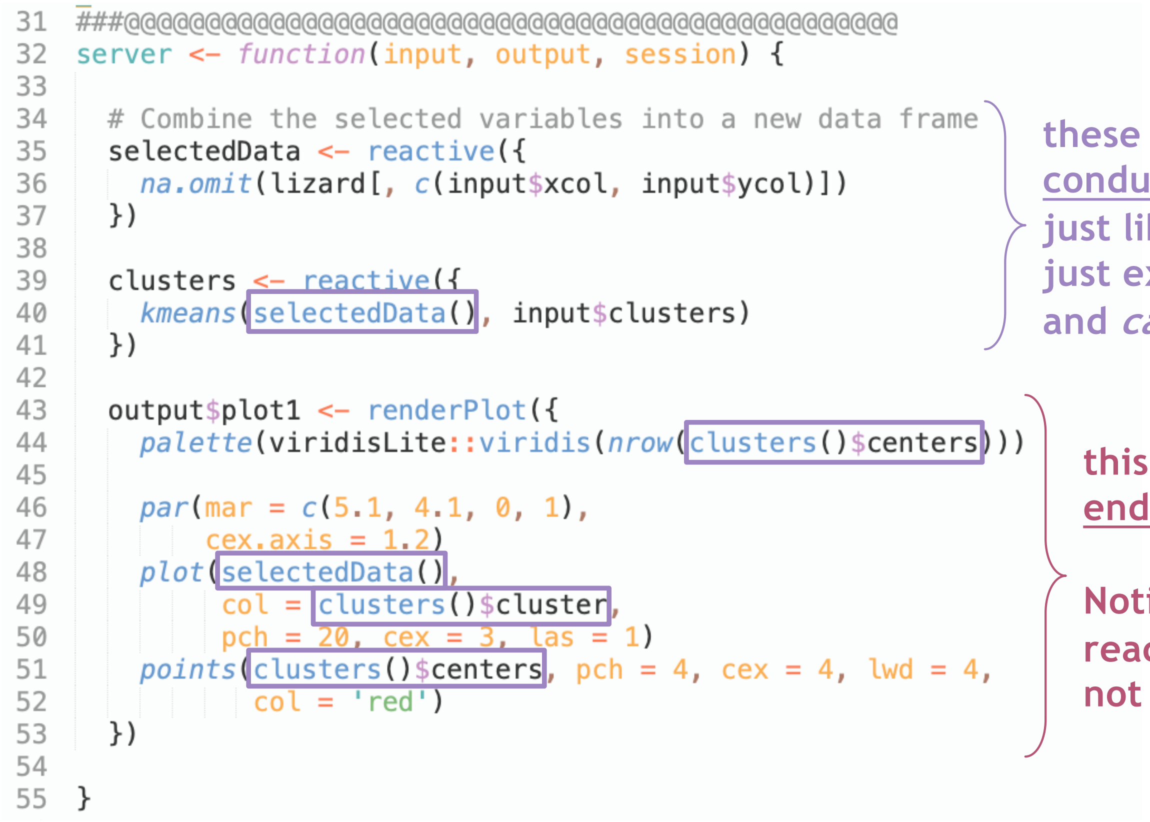
- Nesting multiple reactions within a render is slow (or makes you look like a bot if calling out to a service)



```r
###@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
server <- function(input, output, session) {
  output$plot1 <- renderPlot({
    selectedData <- na.omit(lizard[, c(input$xcol, input$ycol)])
    clusters <- kmeans(selectedData, input$clusters)
    palette(viridisLite::viridis(nrow(clusters$centers)))

    par(mar = c(5.1, 4.1, 0, 1),
        cex.axis = 1.2)
    plot(selectedData,
         col = clusters$cluster,
         pch = 20, cex = 3, las = 1)
    points(clusters$centers, pch = 4, cex = 4, lwd = 4,
           col = 'red')
  })

}
```

All our reactivity is within the render. So any change to input re-renders the whole plot. This example isn't *bad* but bigger applications can quickly ballon this problem

A better server – lizards_betterreact.R



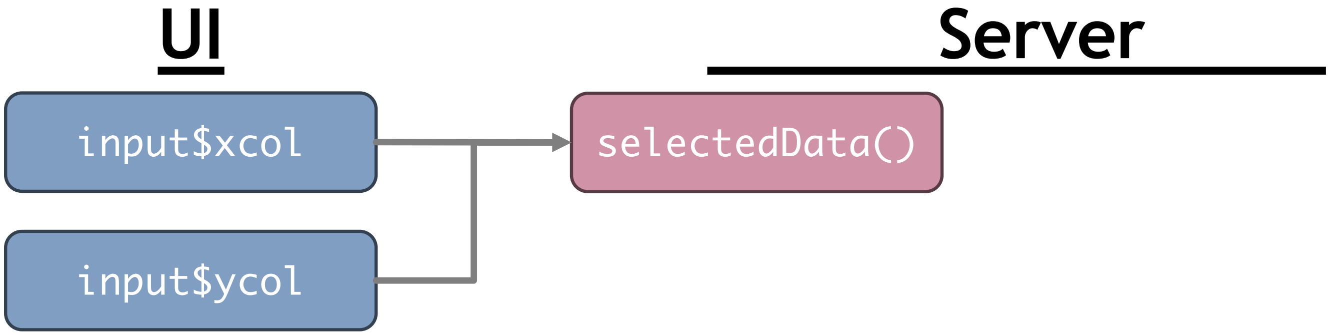
```
###@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
server <- function(input, output, session) {

  # Combine the selected variables into a new data frame
  selectedData <- reactive({
    na.omit(lizard[, c(input$xcol, input$ycol)])
  })

  clusters <- reactive({
    kmeans(selectedData(), input$clusters)
  })

  output$plot1 <- renderPlot({
    palette(viridisLite::viridis(nrow(clusters()$centers)))

    par(mar = c(5.1, 4.1, 0, 1),
        cex.axis = 1.2)
    plot(selectedData(),
         col = clusters()$cluster,
         pch = 20, cex = 3, las = 1)
    points(clusters()$centers, pch = 4, cex = 4, lwd = 4,
           col = 'red')
  })

}
```

these are reactive conductor! they react just like renders do but just execute expressions and *cache values*

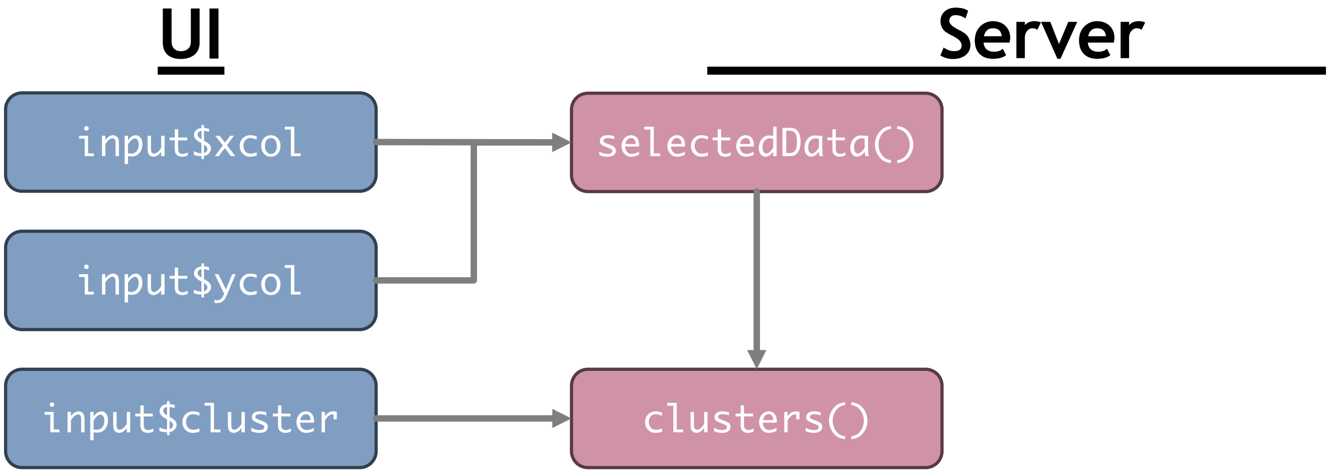
this is also a reactive endpoint!

Notice in this one we react to reactions here not to input

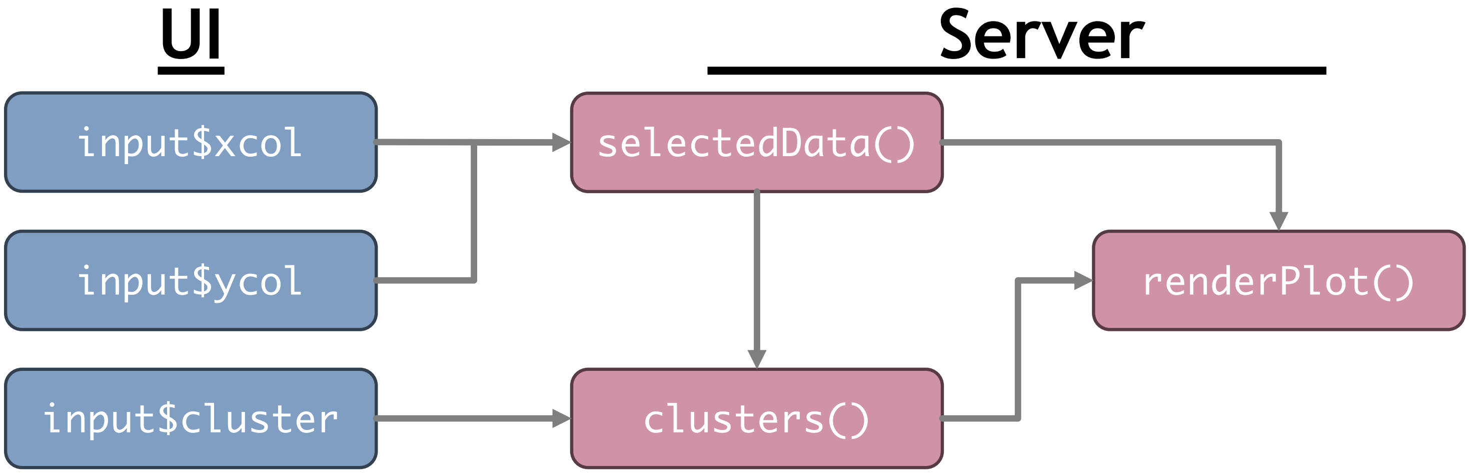
Breaking down that reactivity a bit more…



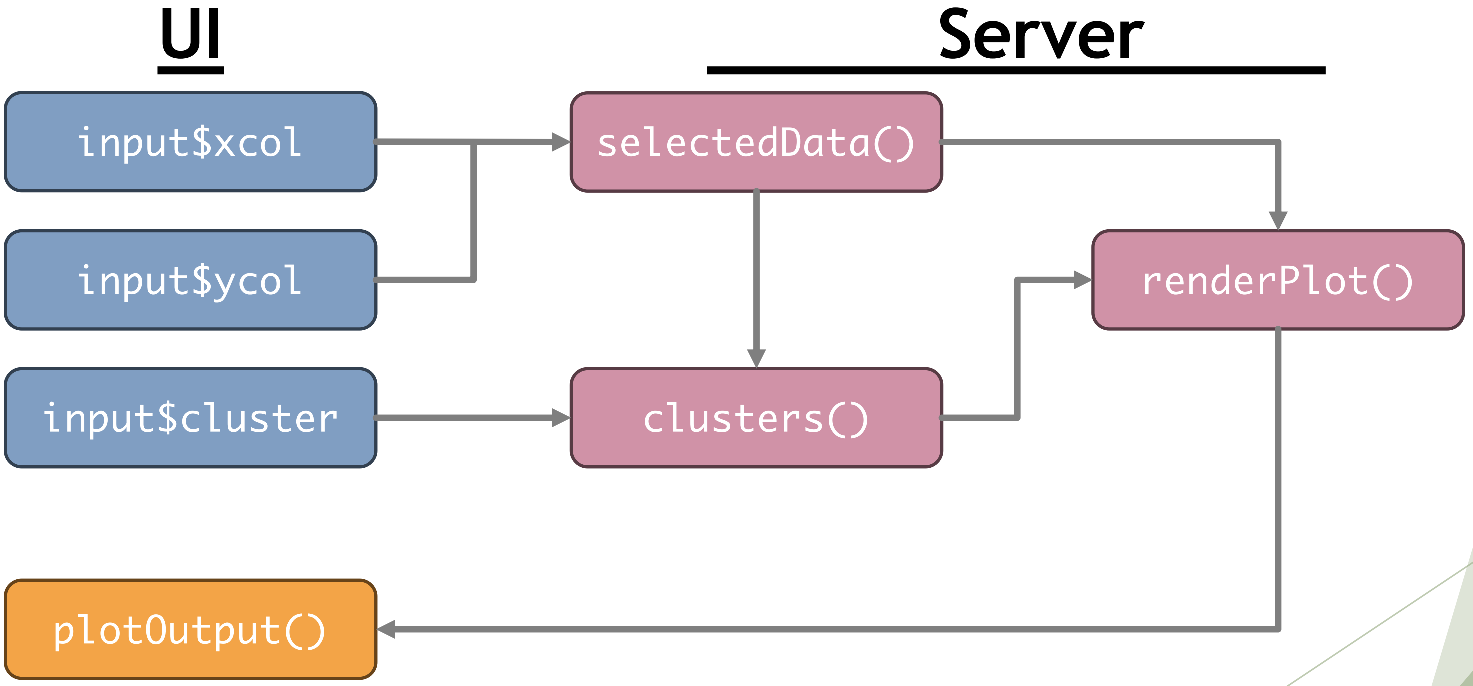
Breaking down that reactivity a bit more…



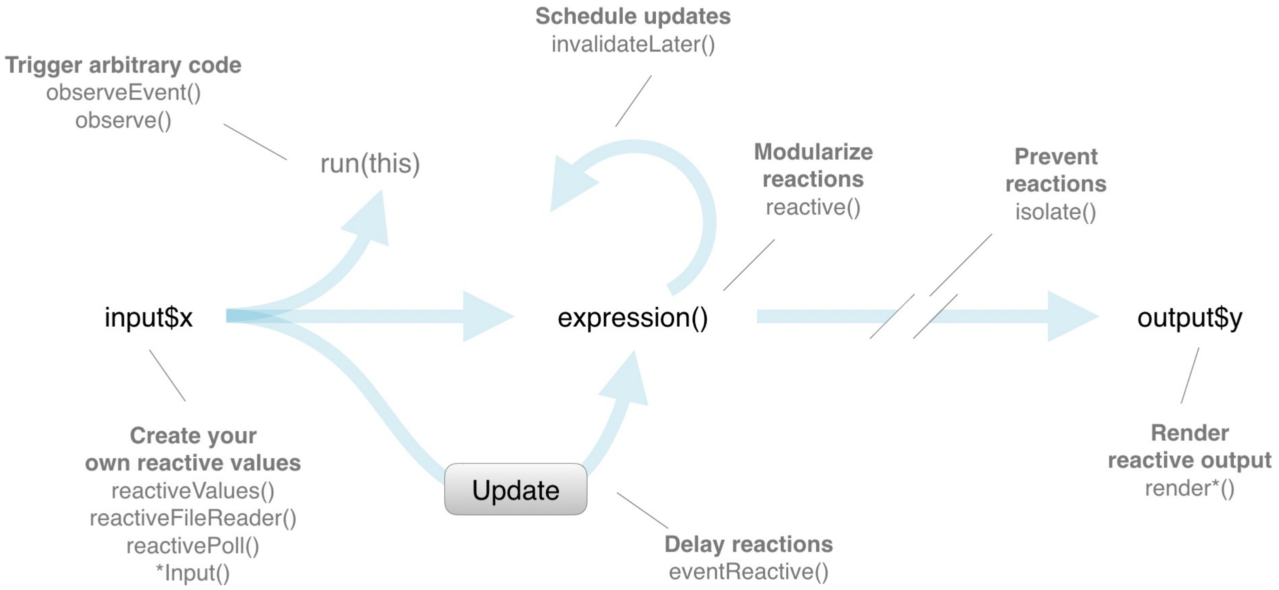
Breaking down that reactivity a bit more…



Breaking down that reactivity a bit more…



Reactivity is a deep well



Schedule updates
invalidateLater()

Trigger arbitrary code
observeEvent()
observe()

run(this)

Modularize reactions
reactive()

Prevent reactions
isolate()

input$x

expression()

output$y

Create your own reactive values
reactiveValues()
reactiveFileReader()
reactivePoll()
*Input()

Update

Delay reactions
eventReactive()

Render reactive output
render*()

We will revisit this advanced reactivity on day 3

5 minute break

Resources

- https://shiny.posit.co/r/getstarted/shiny-basics/lesson1/
- https://mastering-shiny.org/index.html
- https://github.com/nanxstats/awesome-shiny-extensions