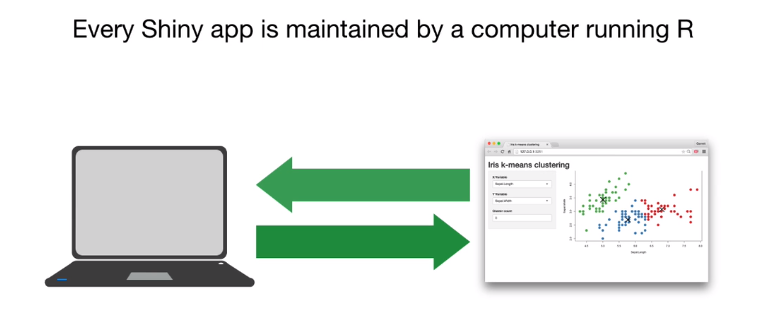
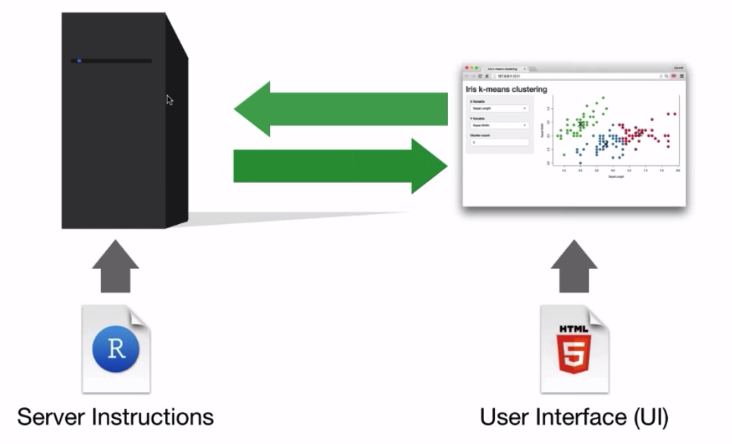
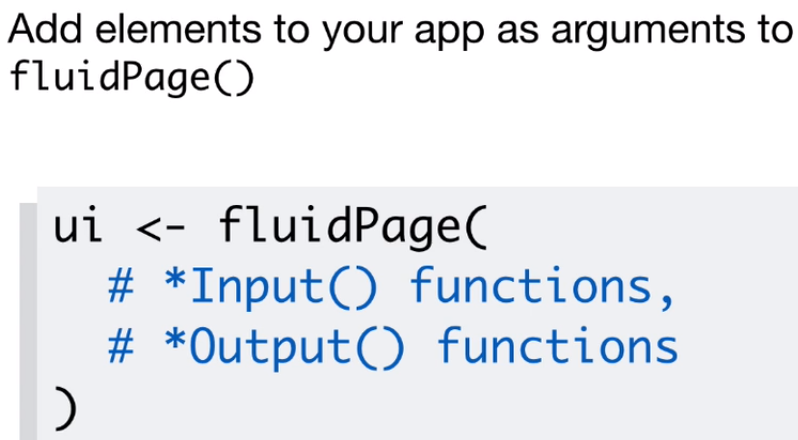
**RSHINY:**

Link: <https://vimeo.com/rstudioinc/review/131218530/212d8a5a7a/#t=0m0s>

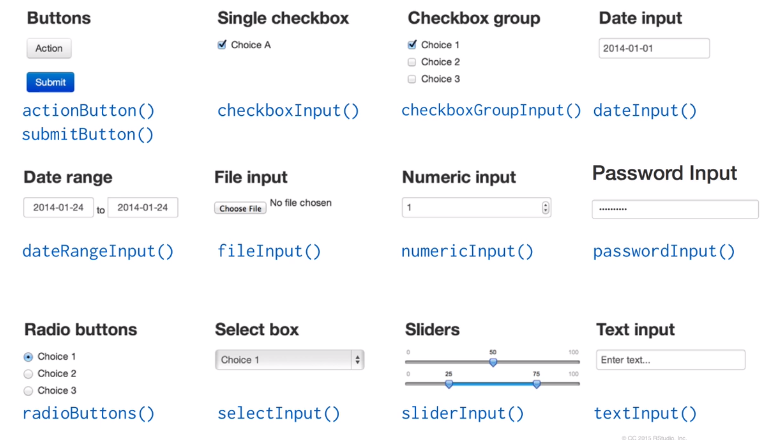




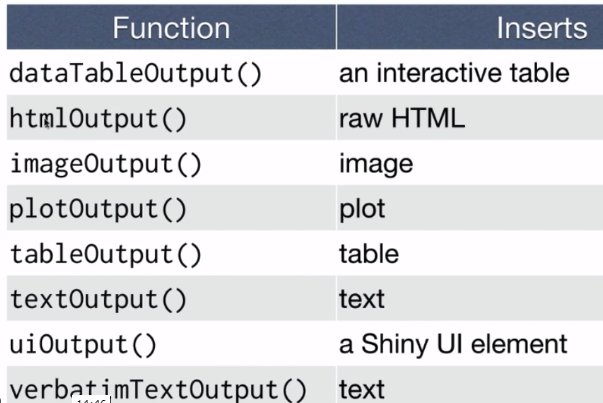




Input functions:

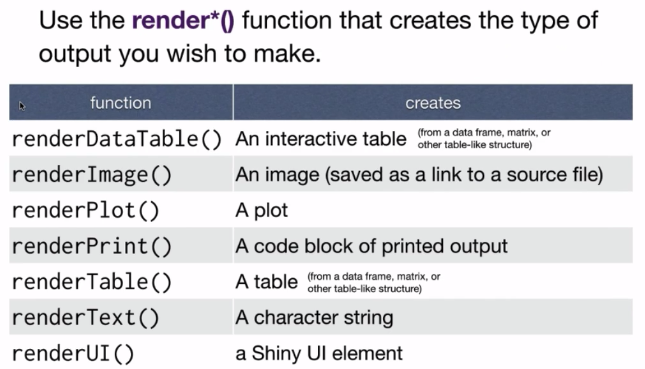


Output functions:

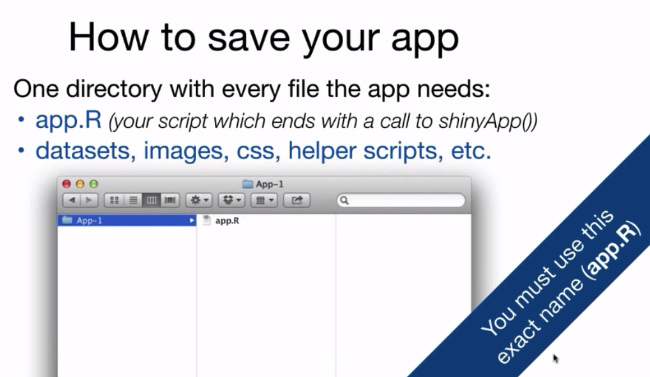


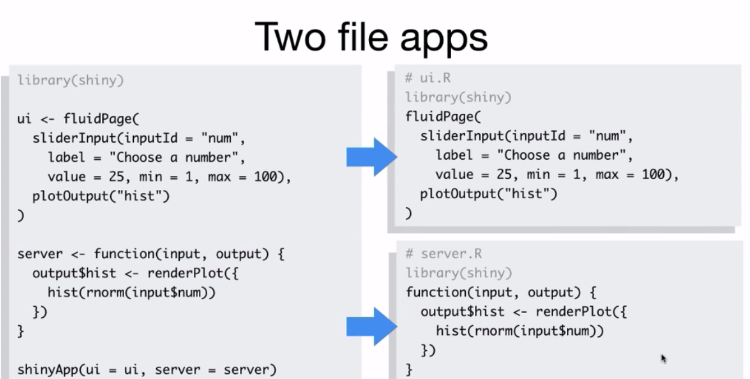
3 rules to write server function:

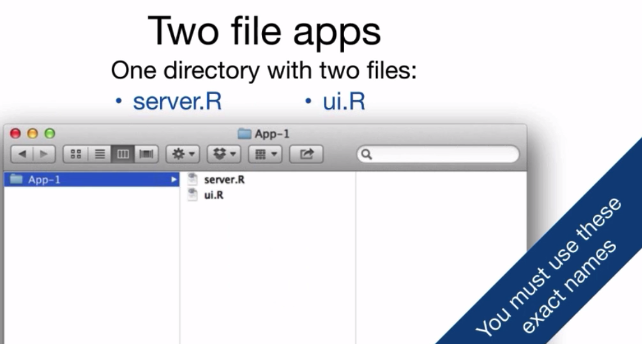
1. Save objects to display to output$
2. Build objects to display with render\*()

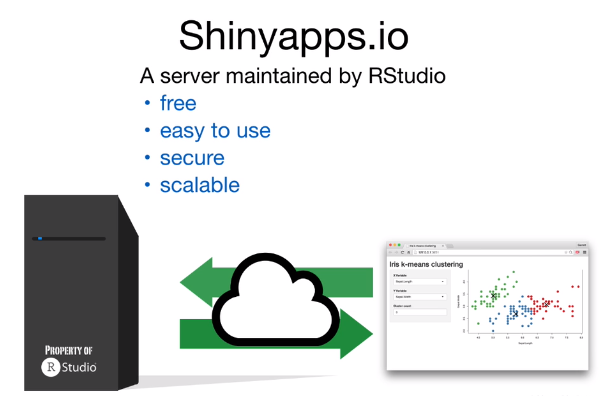


1. Use input values with input$









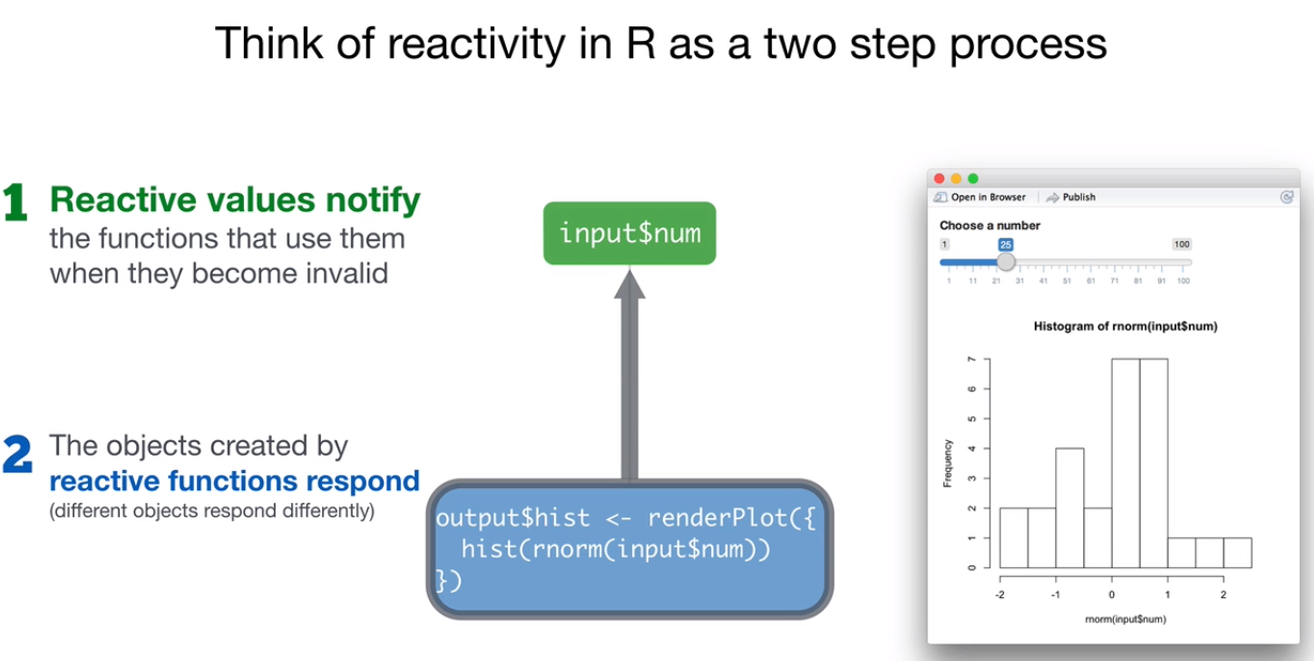
Hosting dashboard in shineyapps.io

<http://shiny.rstudio.com/articles/shinyapps.html>

Shiny Server – A back end program that builds a linux web server specifically designed to host shiny apps.

[www.rstudio.com/products/shiny/shiny-server/](http://www.rstudio.com/products/shiny/shiny-server/) (Free)

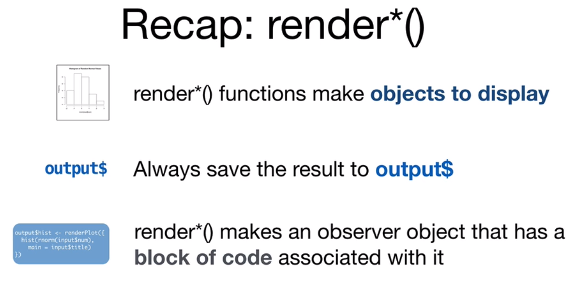
Shiny Server pro is for commercial use (chargeable)



Step 1 is also called as invalidating process

Reactive toolkit: (7 steps)

1. render\*({ })
   1. builds reactive output to display in UI
   2. Use a code chunk to build (and rebuild) and object
      1. What code will the function use?
   3. The object will respond to changes in a asset of reactive values
      1. Which reactive values will the object respond to?



Eg:

library(shiny)

ui <- fluidPage(

sliderInput(inputId = "num", label="choose a number",

value=25, min=1, max=100),

textInput(inputId="title", label="Write a title", value ="Histogram of Random Normal Values"),

plotOutput("hist"),

verbatimTextOutput("stats")

)

# how to assemble inputs to outputs

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

output$hist <- renderPlot({

hist(rnorm(input$num), main=input$title)

})

output$stats <- renderPrint({

summary(rnorm(input$num))

})

}

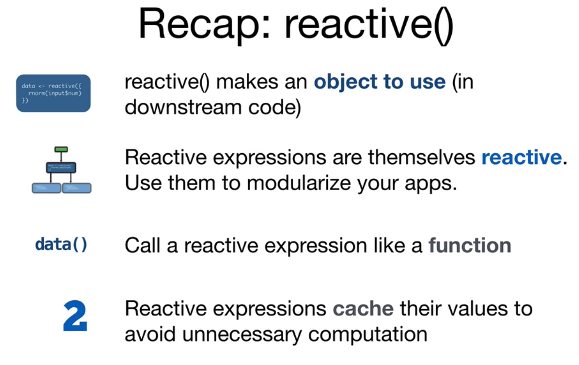
shinyApp(ui = ui, server = server)

**Modularize code with reactive()**

1. reactive()

reactive expression is special in 2 ways

* 1. you call a reactive expression like a function - data()
  2. reactive expression caches the latest value (The expression will return its most recent value, unless it has become invalidated)



example

library(shiny)

ui <- fluidPage(

sliderInput(inputId="num", label="Enter a number",

value = 25, min=1, max=100),

plotOutput("hist"),

verbatimTextOutput("stats")

)

##Building once and accessing twice

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

data <- reactive({

rnorm(input$num)

})

output$hist <- renderPlot({

hist(data())

})

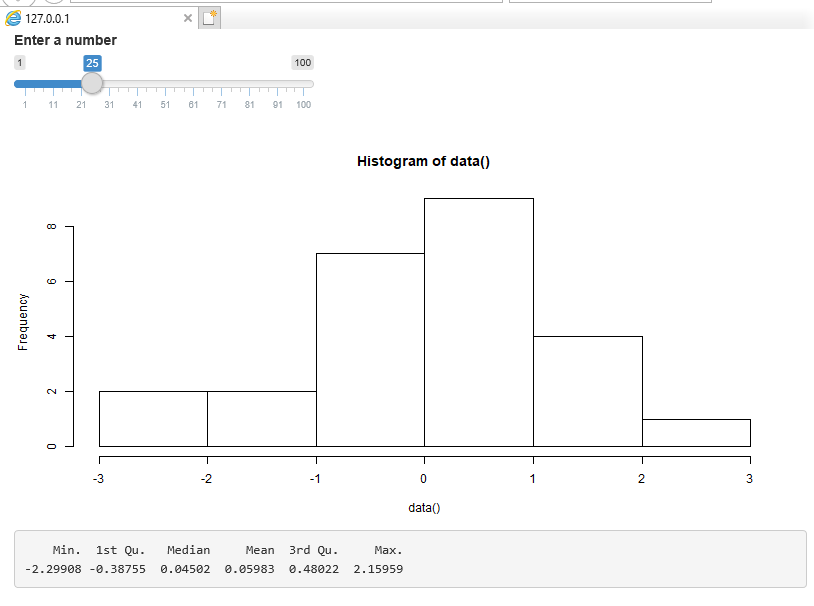
output$stats <- renderPrint({

summary(data())

})

}

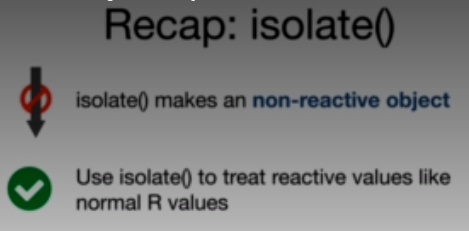
shinyApp(ui, server)



1. isolate()

eg: preventing the title field from updating the plot while editing.

Returns the result as a non-reactive value.



library(shiny)

ui <- fluidPage(

sliderInput(inputId = "num", label = "Write a num: ", value=25, min=1, max=100),

textInput(inputId = "title", label ="Give a title:", value="Histogram"),

plotOutput("hist")

)

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

output$hist <- renderPlot({

hist(rnorm(input$num), main = isolate({input$title}) )

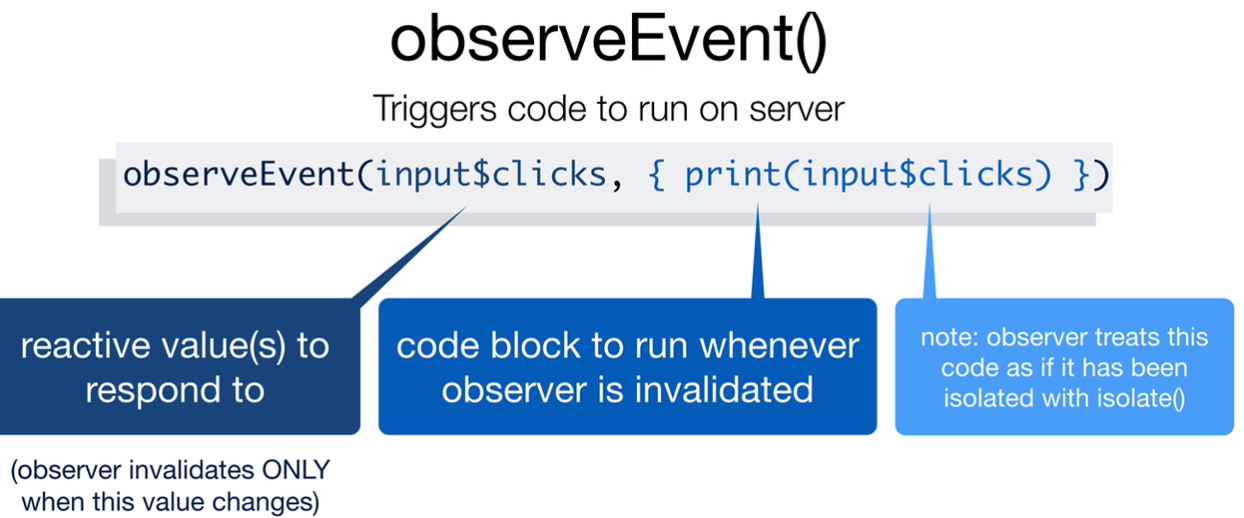
})

}

shinyApp(ui, server)

O/p: This will not make the title change while we are typing instead once the plot changes, it changes the title in a non-reactive way.

1. Trigger code with observeEvent()
   1. Triggers code to run on server



library(shiny)

ui <- fluidPage(

actionButton(inputId = "go", label="Click me!")

)

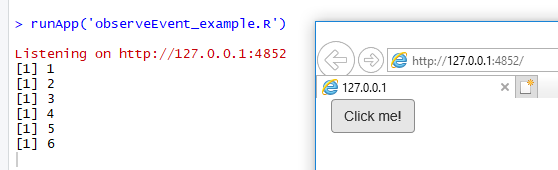
server <- function(input, output, session) {

observeEvent(input$go, { print(as.numeric(input$go))})

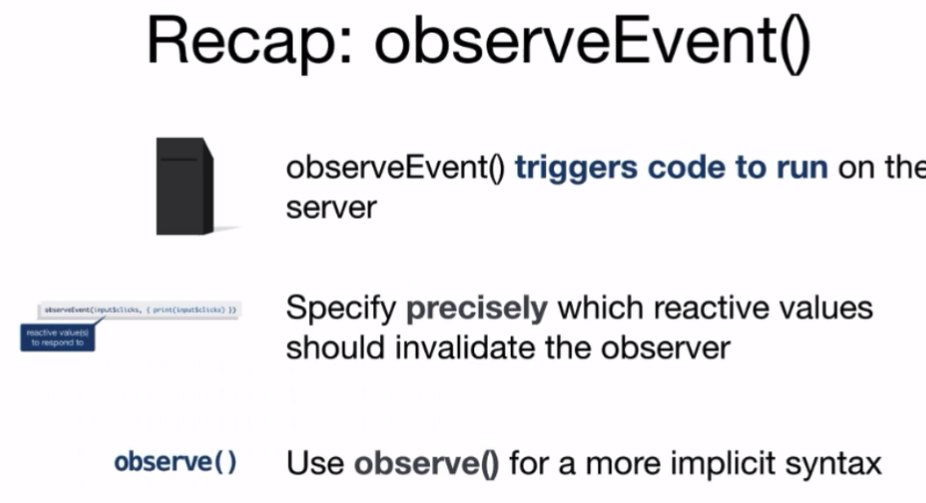
}

shinyApp(ui, server)

o/p: whenever button is clicked, value prints in console.



1. observe()
   1. similar to reactive
   2. difference is it tracks every reactive value , instead of focusing on one like observeEvent()



1. Delay reactions with eventReactive()

Can prevent the graph from updating until we hit the button.

library(shiny)

ui <- fluidPage(

sliderInput(inputId="num", label="choose a number:",

value=25, min=1, max=100),

actionButton(inputId="go", label="Update"),

plotOutput("hist")

)

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

data <- eventReactive(input$go, {rnorm(input$num)})

output$hist <- renderPlot({

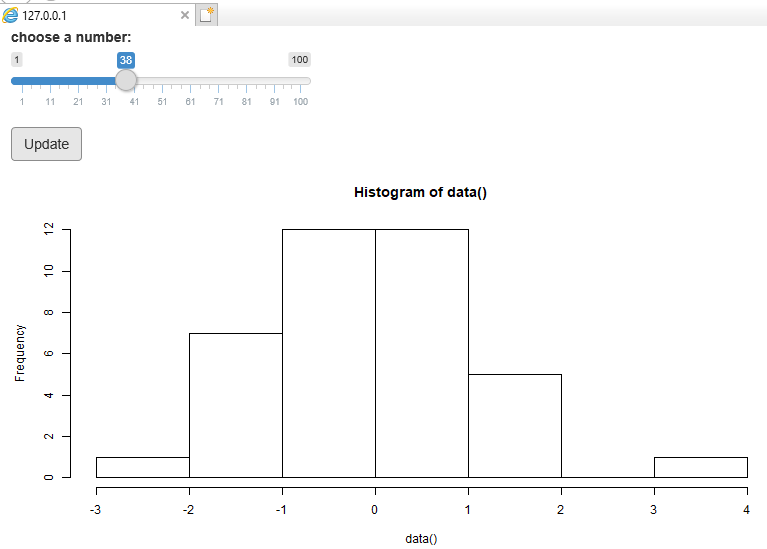
hist(data())

})

}

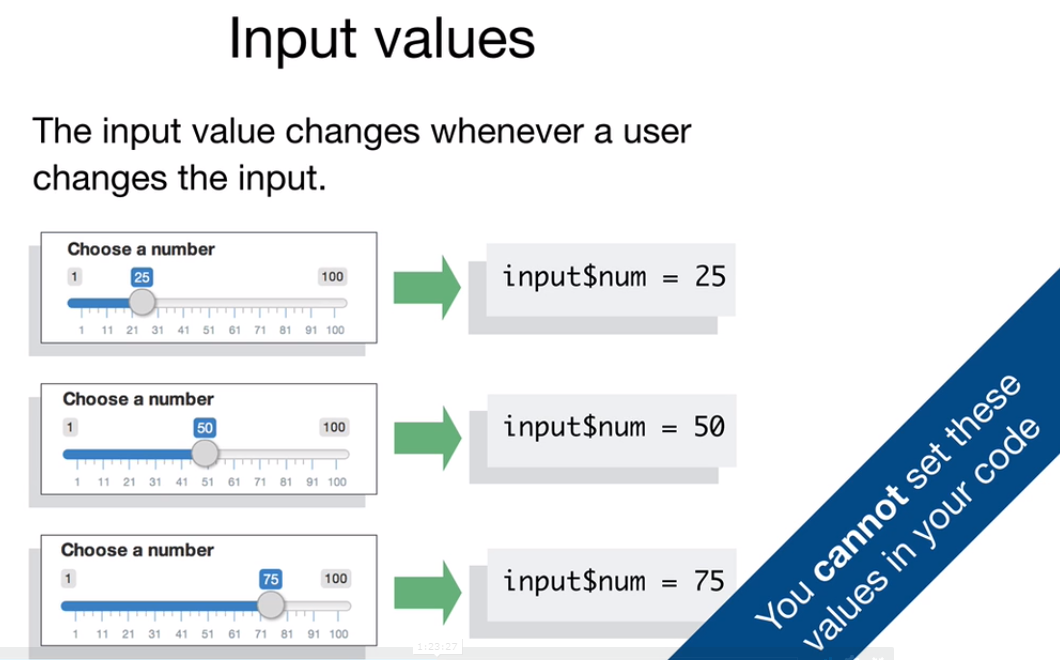
shinyApp(ui, server)

o/p: only when update button is clicked plot changes.





1. Manage state with reactiveValues()
   1. Creates a list of reactive values to manipulate programmatically
   2. Rv <- reactiveValues(data=rnorm(100))



library(shiny)

ui <- fluidPage(

actionButton(inputId="norm", label="Normal"),

actionButton(inputId="unif", label="uniform"),

plotOutput("hist")

)

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

rv <- reactiveValues(data =rnorm(100))

observeEvent(input$norm, {rv$data <- rnorm(100)})

observeEvent(input$unif , {rv$data <- runif(100)})

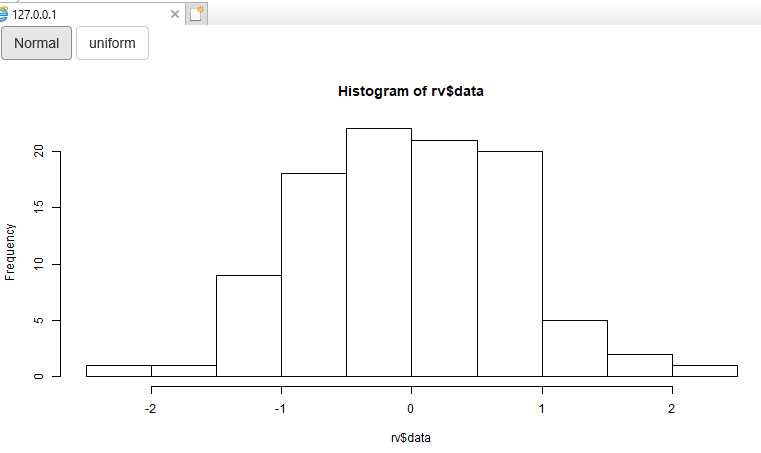
output$hist <- renderPlot({

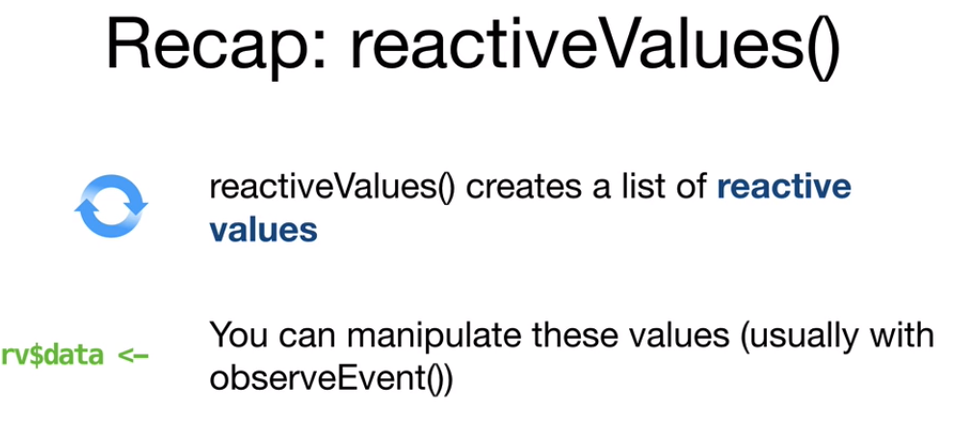
hist(rv$data)

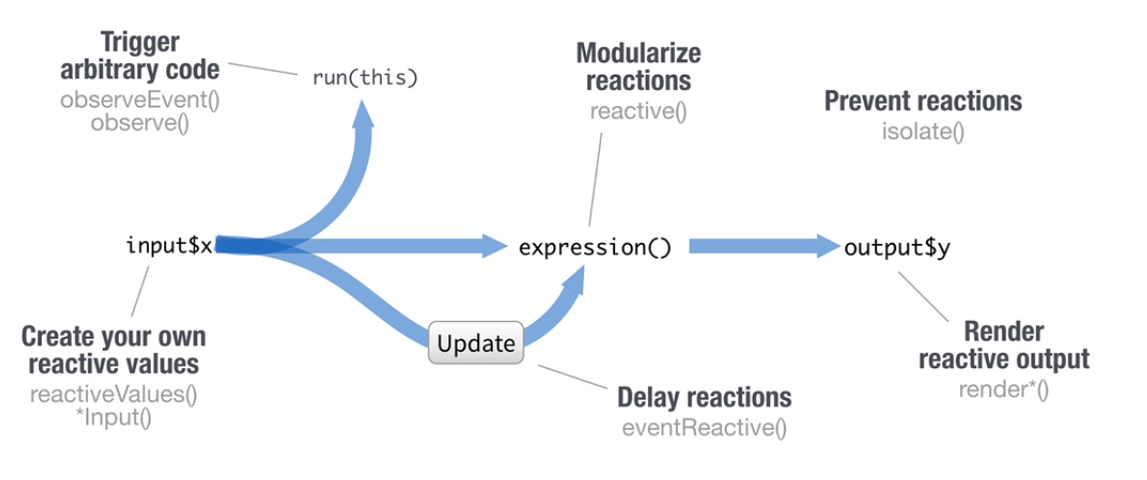
})

}

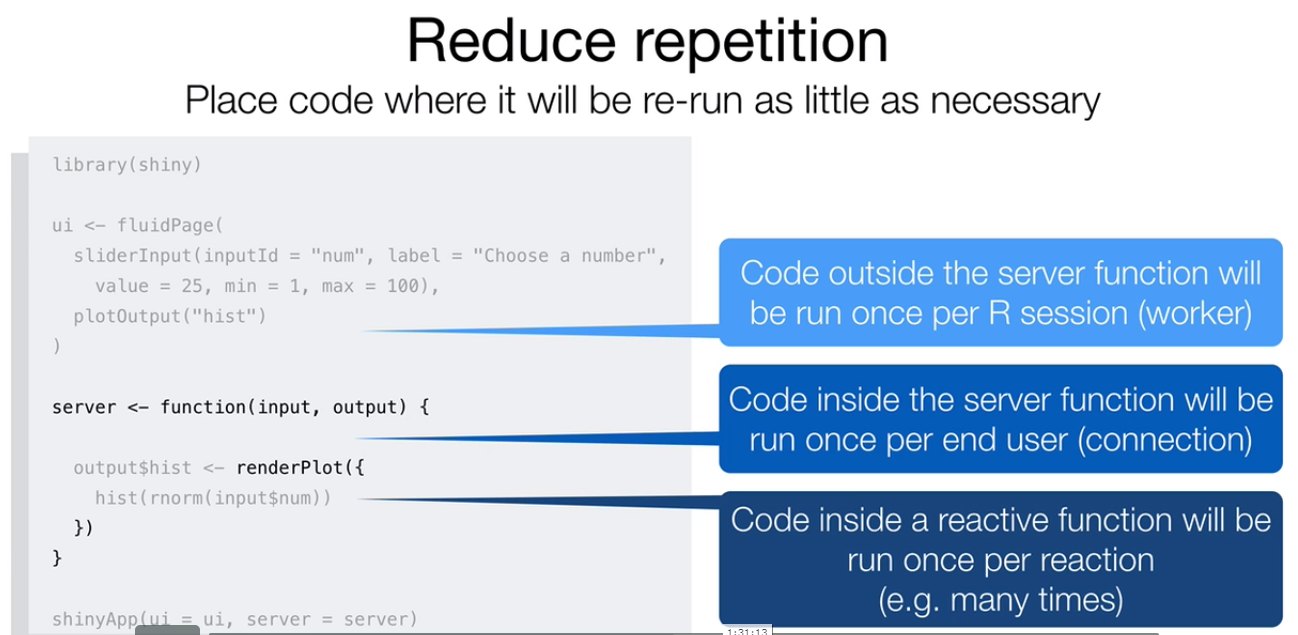
shinyApp(ui, server)







Parting tips:



1. How can R possibly implement reactivity?

<http://shiny.rstudio.com/articles/understanding-reactivity.html>