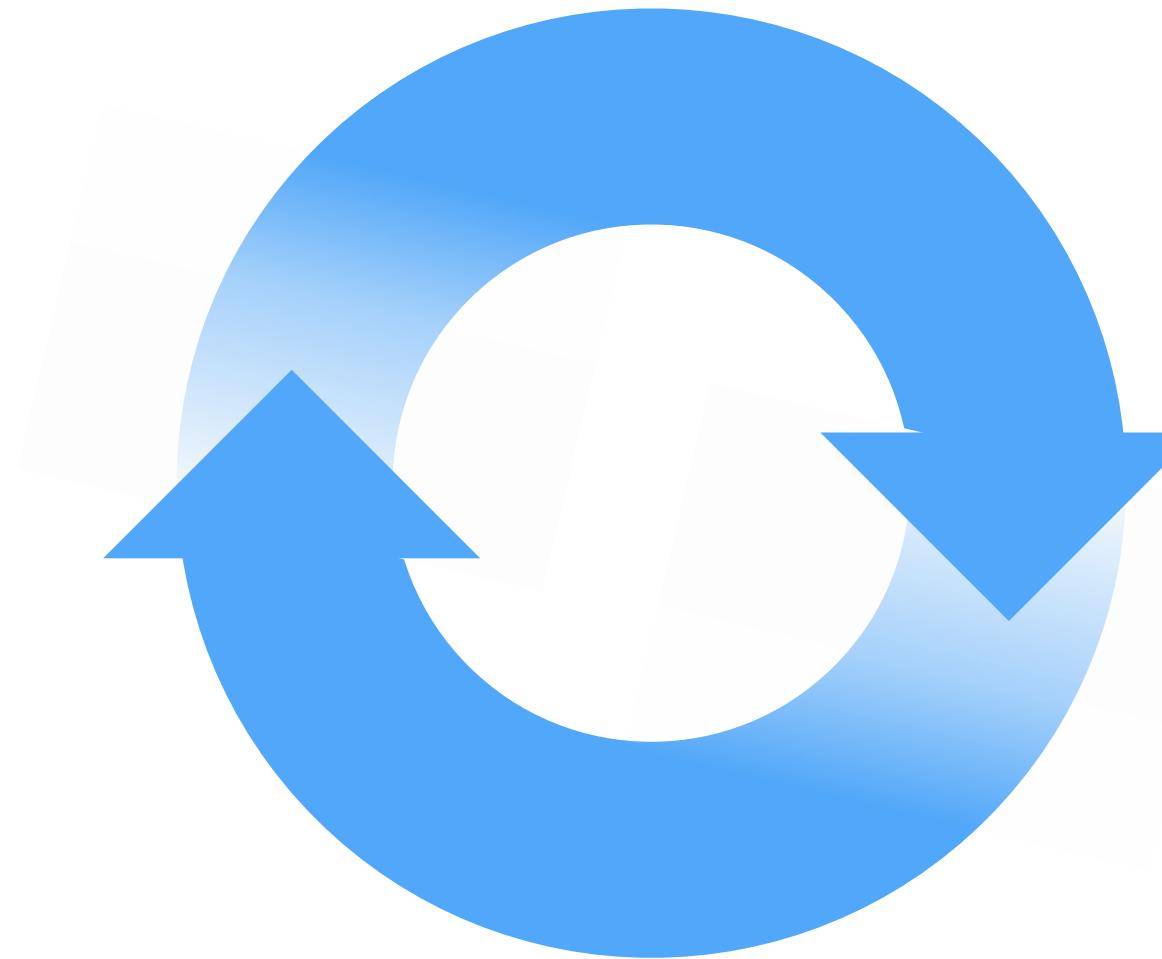


All Training materials are provided "as is" and without warranty and RStudio disclaims any and all express and implied warranties including without limitation the implied warranties of title, fitness for a particular purpose, merchantability and noninfringement.

The Training Materials are licensed under the Creative Commons Attribution-Noncommercial 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/3.0/us/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

How to start with Shiny, Part 2

How to customize reactions



Garrett Grolemund

Data Scientist and Master Instructor

May 2015

Email: garrett@rstudio.com

Twitter: @StatGarrett

Code and slides at:

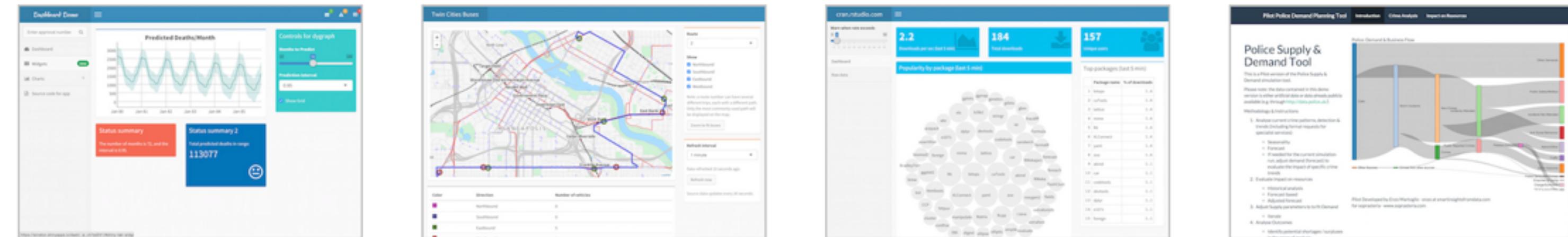
bit.ly/shiny-quickstart-2



Shiny Showcase

www.rstudio.com/products/shiny/shiny-user-showcase/

Shiny Apps for the Enterprise



Shiny Dashboard Demo

A dashboard built with Shiny.

Location tracker

Track locations over time with streaming data.

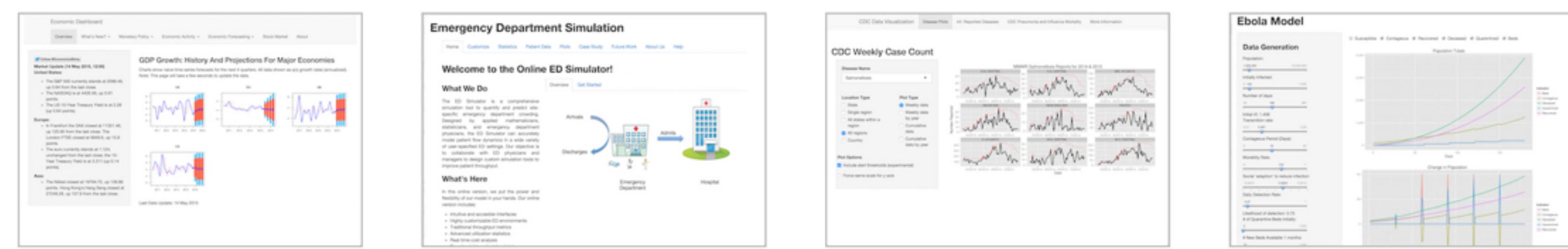
Download monitor

Streaming download rates visualized as a bubble chart.

Supply and Demand

Forecast demand to plan resource allocation.

Industry Specific Shiny Apps



Economic Dashboard

Economic forecasting with macroeconomic indicators.

ER Optimization

An app that models patient flow.

CDC Disease Monitor

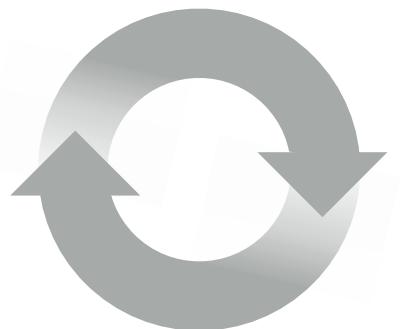
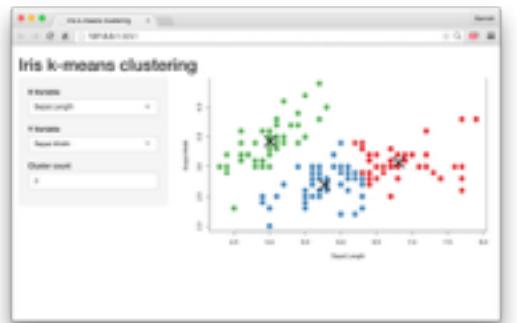
Alert thresholds and automatic weekly updates.

Ebola Model

An epidemiological simulation.



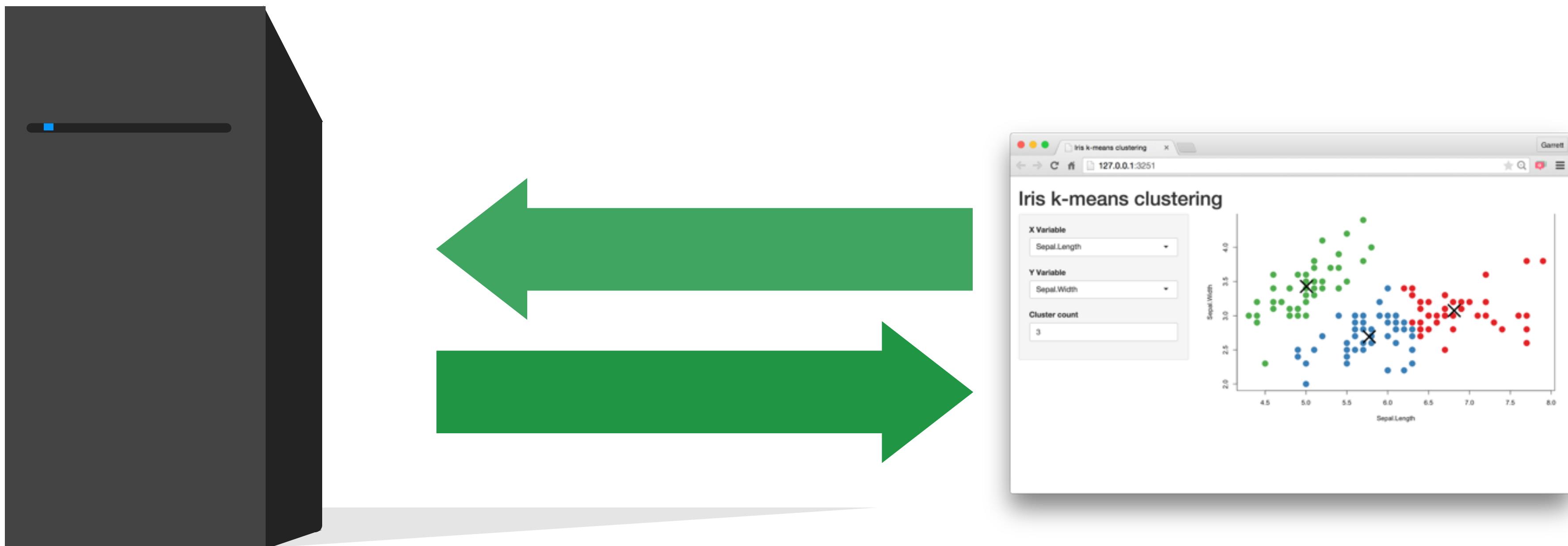
How to start with Shiny



1. How to build a Shiny app (www.rstudio.com/resources/webinars/)
2. How to customize reactions (Today)
3. How to customize appearance (June 3)

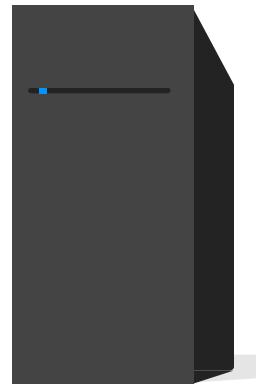
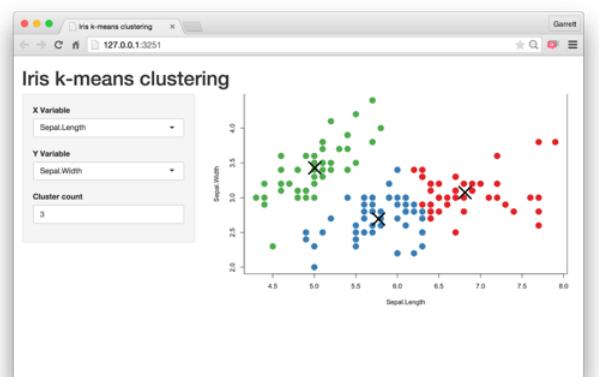
**The story
so far**

Every Shiny app is maintained by a computer running R



App template

The shortest viable shiny app



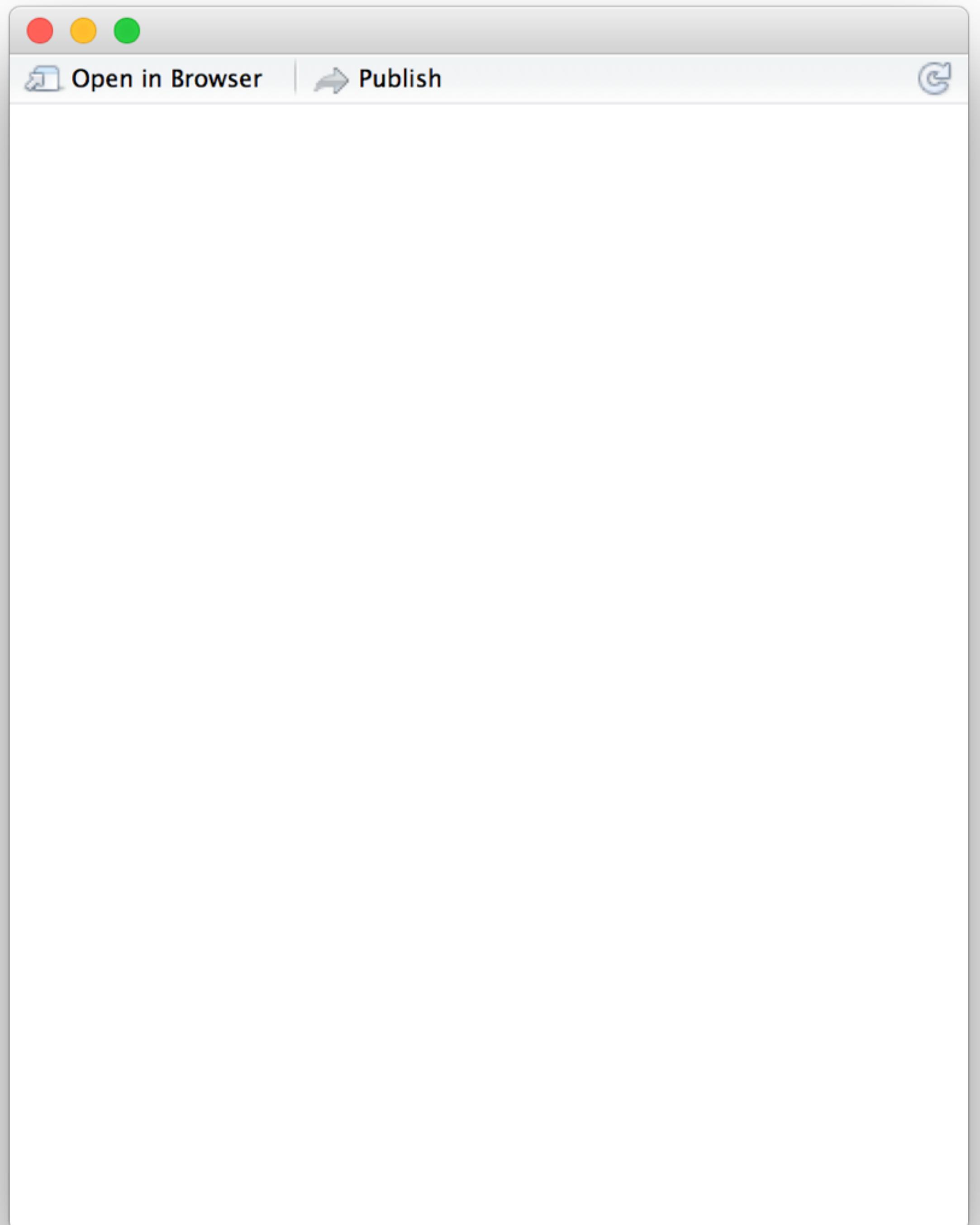
```
library(shiny)  
ui <- fluidPage()  
  
server <- function(input, output) {}  
  
shinyApp(ui = ui, server = server)
```

```
library(shiny)

ui <- fluidPage(
  )

server <- function(input, output) {
  }

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



```
library(shiny)

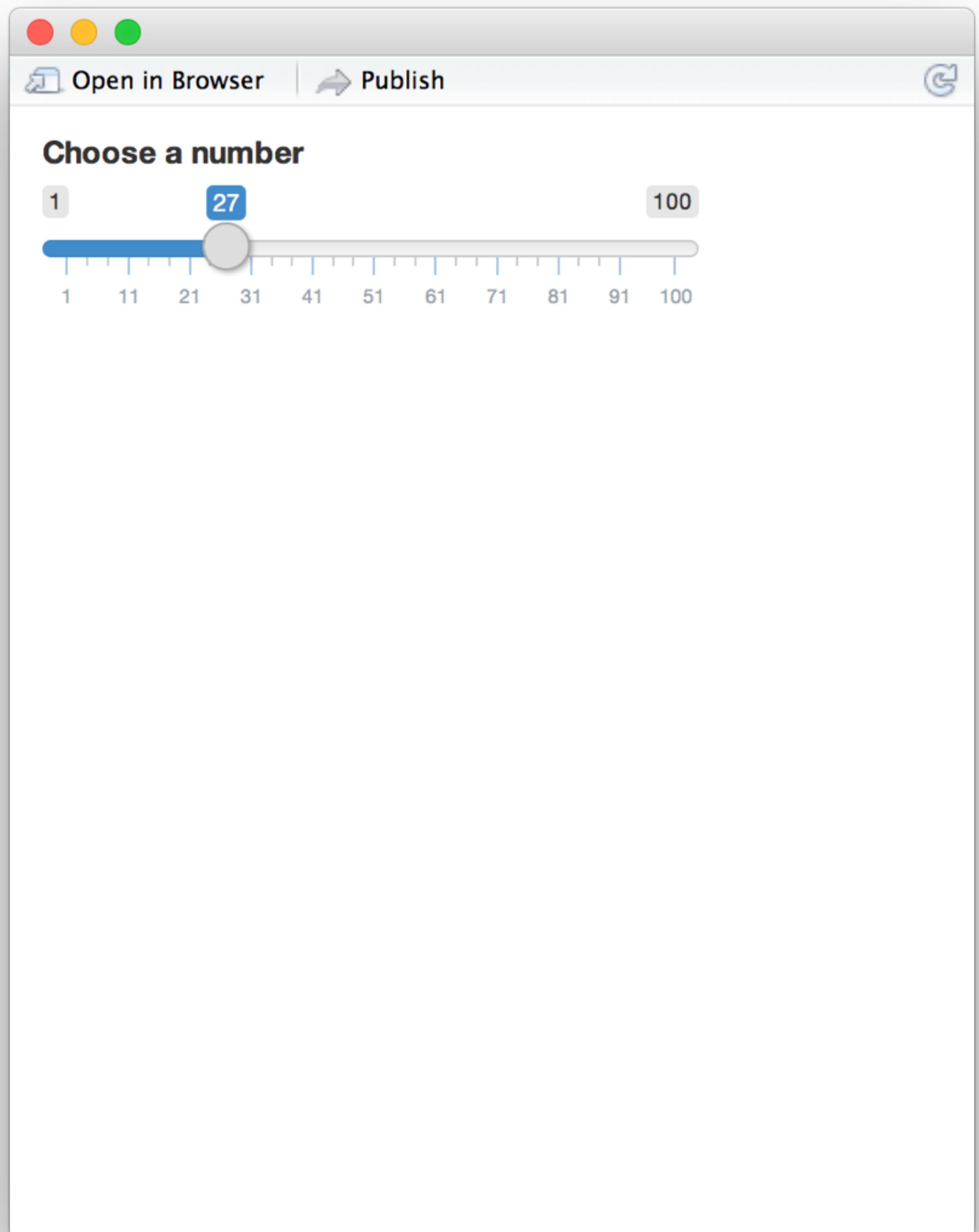
ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100)

)

server <- function(input, output) {

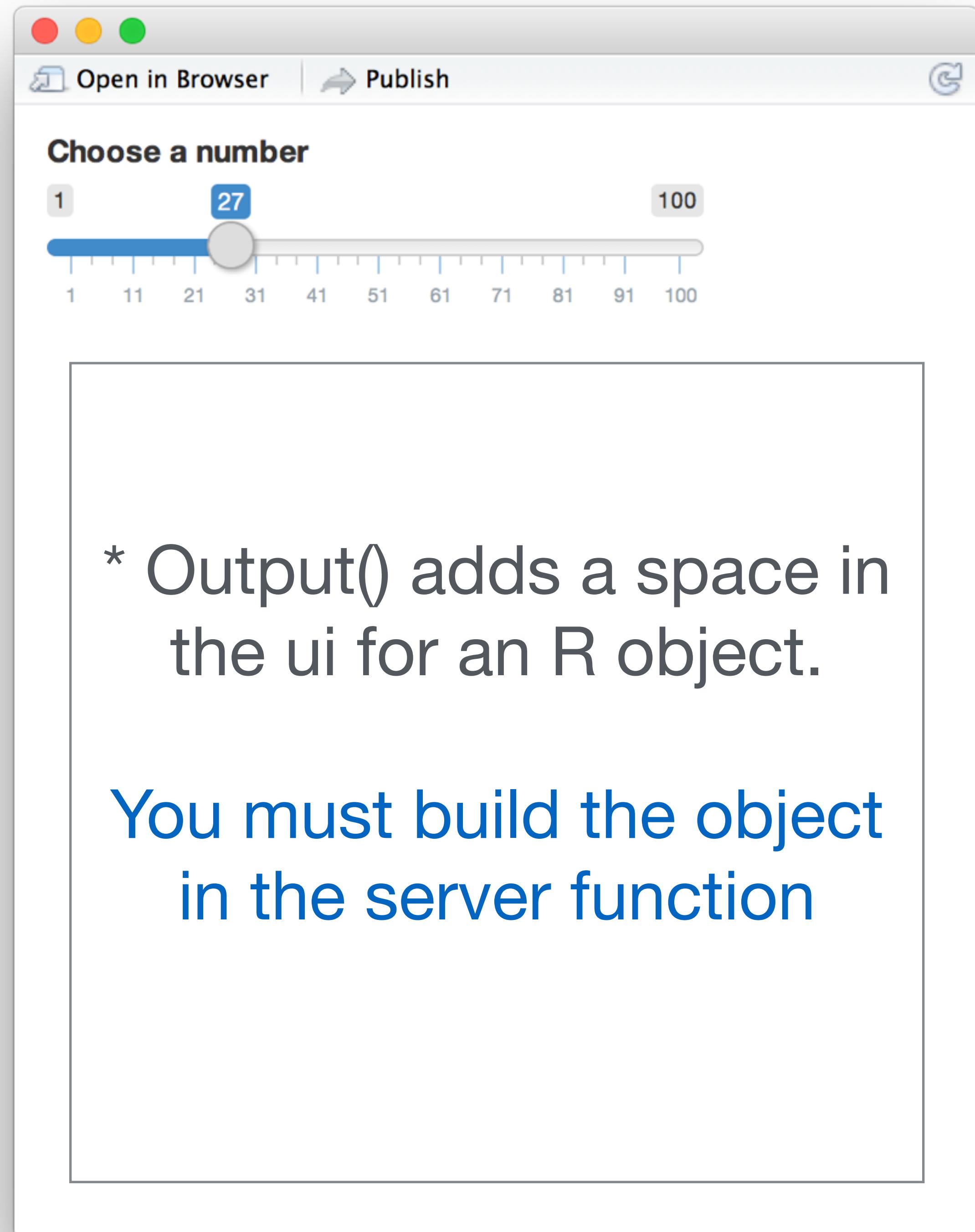
}

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



```
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)
server <- function(input, output) {
}
shinyApp(ui = ui, server = server)
```



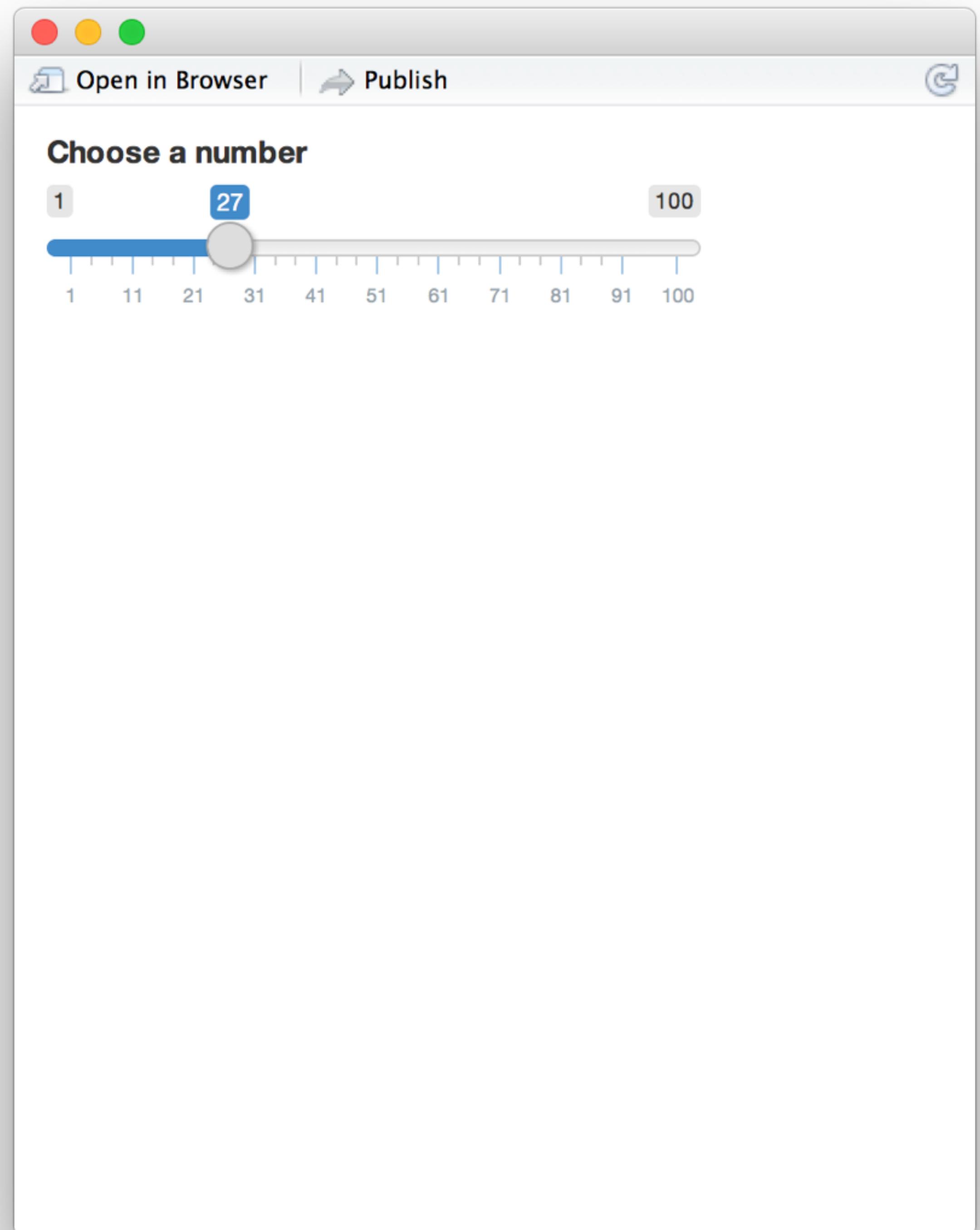
```
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)

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

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

1



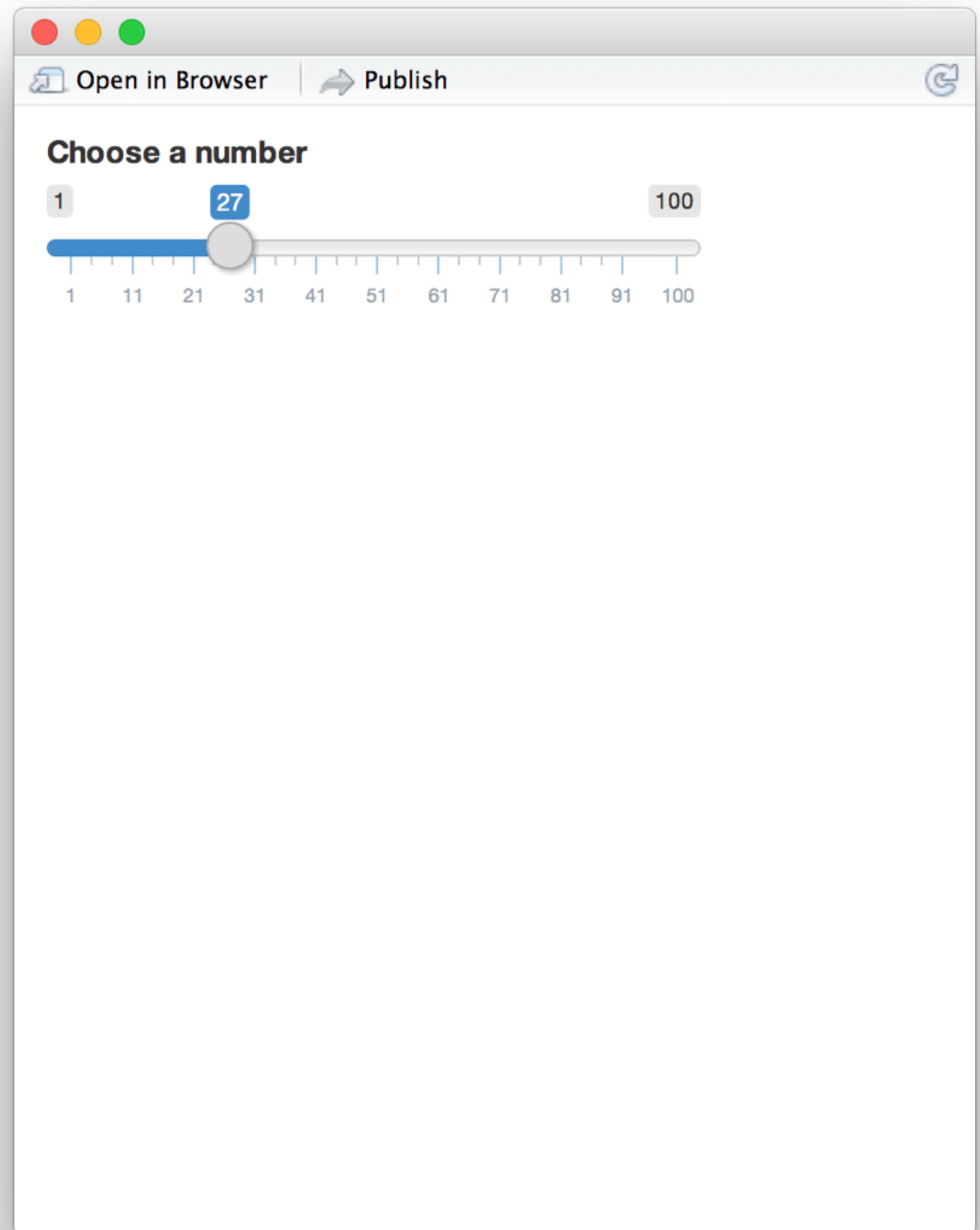
```
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)

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

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

2



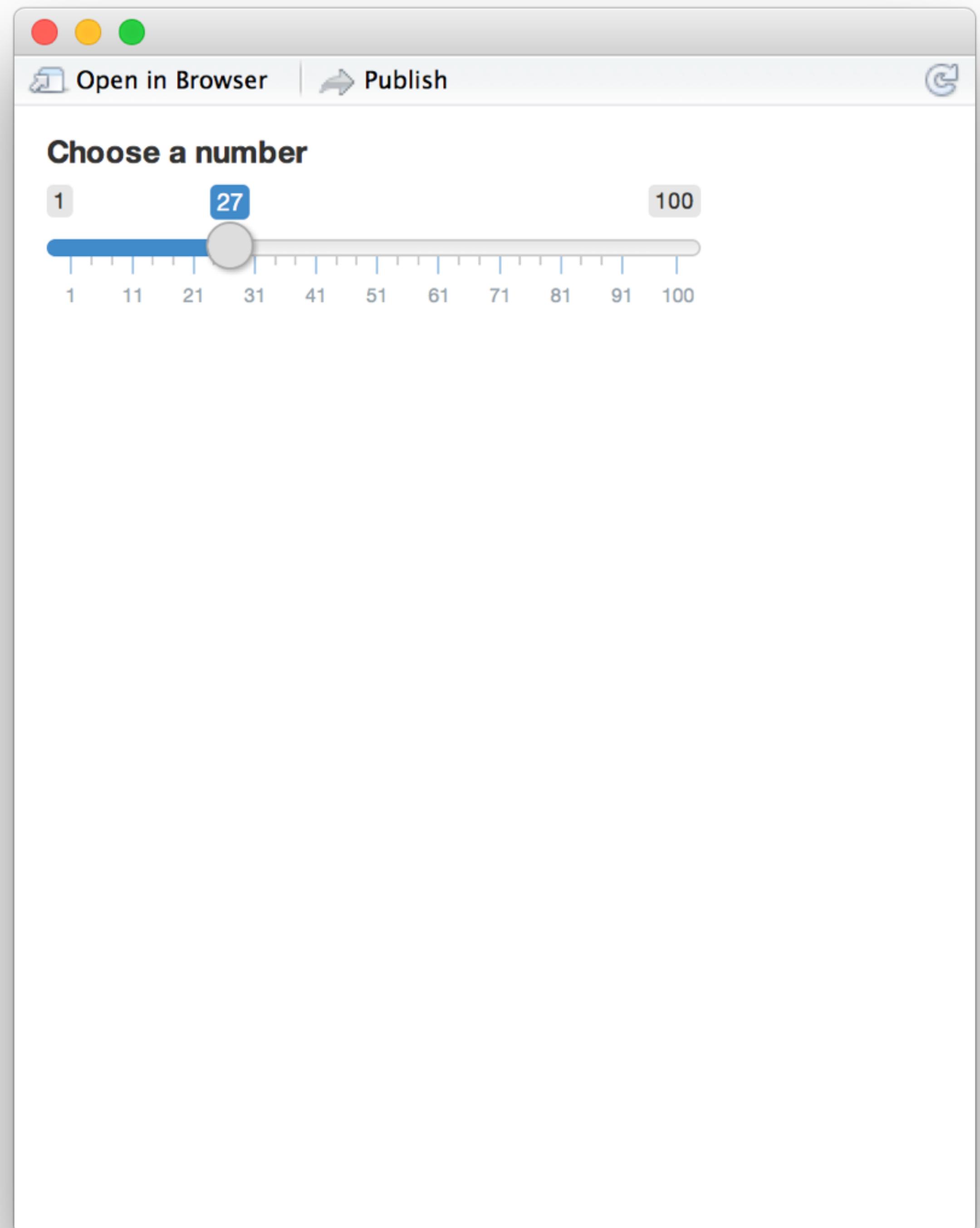
```
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)

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

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

3

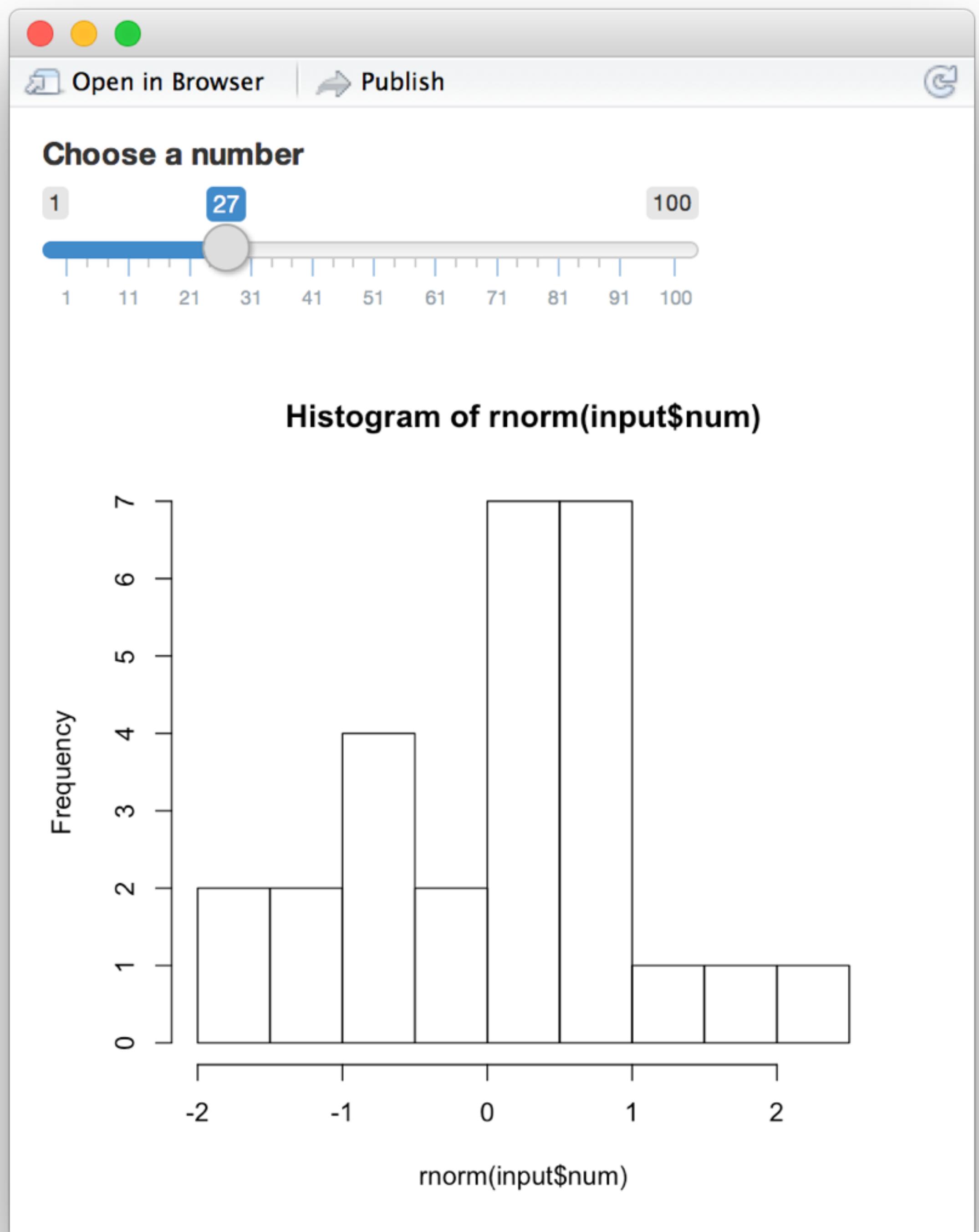


```
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)

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

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



Sharing apps



Shiny Server (Pro)

<http://www.rstudio.com/resources/webinars/>

**what is
Reactivity?**

Think Excel.

The screenshot shows a Microsoft Excel application window titled "Workbook1". The ribbon menu is visible at the top, featuring tabs for Home, Layout, Tables, Charts, SmartArt, Formulas, Data, and Review. The Home tab is selected, displaying various tools for editing, font, alignment, number formats, and styles. A blank worksheet is open, with the active cell being A1. The column headers (A through Q) and row headers (1 through 25) are visible along the edges of the grid. The status bar at the bottom indicates "Normal View" and "Ready".

Workbook1

Tables Charts SmartArt Formulas Data Review

Font Alignment Number Format Cells

Body (Body) 12 A A abc Wrap Text General % , .00 .00 Conditional Formatting Styles Insert Delete

I U Merge

C D E F G H I J K L M N O

50				= F4 + 1
----	--	--	--	----------

© CC 2015 RStudio, Inc.

Workbook1

Tables Charts SmartArt Formulas Data Review

Font Alignment Number Format Cells

Body (Body) 12 A A abc Wrap Text General % , .00 .00 Conditional Formatting Styles Insert Delete

I U A Merge

C D E F G H I J K L M N O

50 51

© CC 2015 RStudio, Inc.

Workbook1

Tables Charts SmartArt Formulas Data Review

Font Alignment Number Format Cells

Body (Body) 12 A A abc Wrap Text General % , .00 .00 Conditional Formatting Styles Insert Delete

I U Merge

C D E F G H I J K L M N O

100 101

Workbook1

Tables Charts SmartArt Formulas Data Review

Font Alignment Number Format Cells

Body (Body) 12 A A abc Wrap Text General % , .00 .00 Conditional Formatting Styles Insert Delete

I U Merge

C D E F G H I J K L M N O

999 1000

© CC 2015 RStudio, Inc.

Workbook1

Tables Charts SmartArt Formulas Data Review

Font Alignment Number Format Cells

1000 → 1001

C D E F G H I J K L M N O

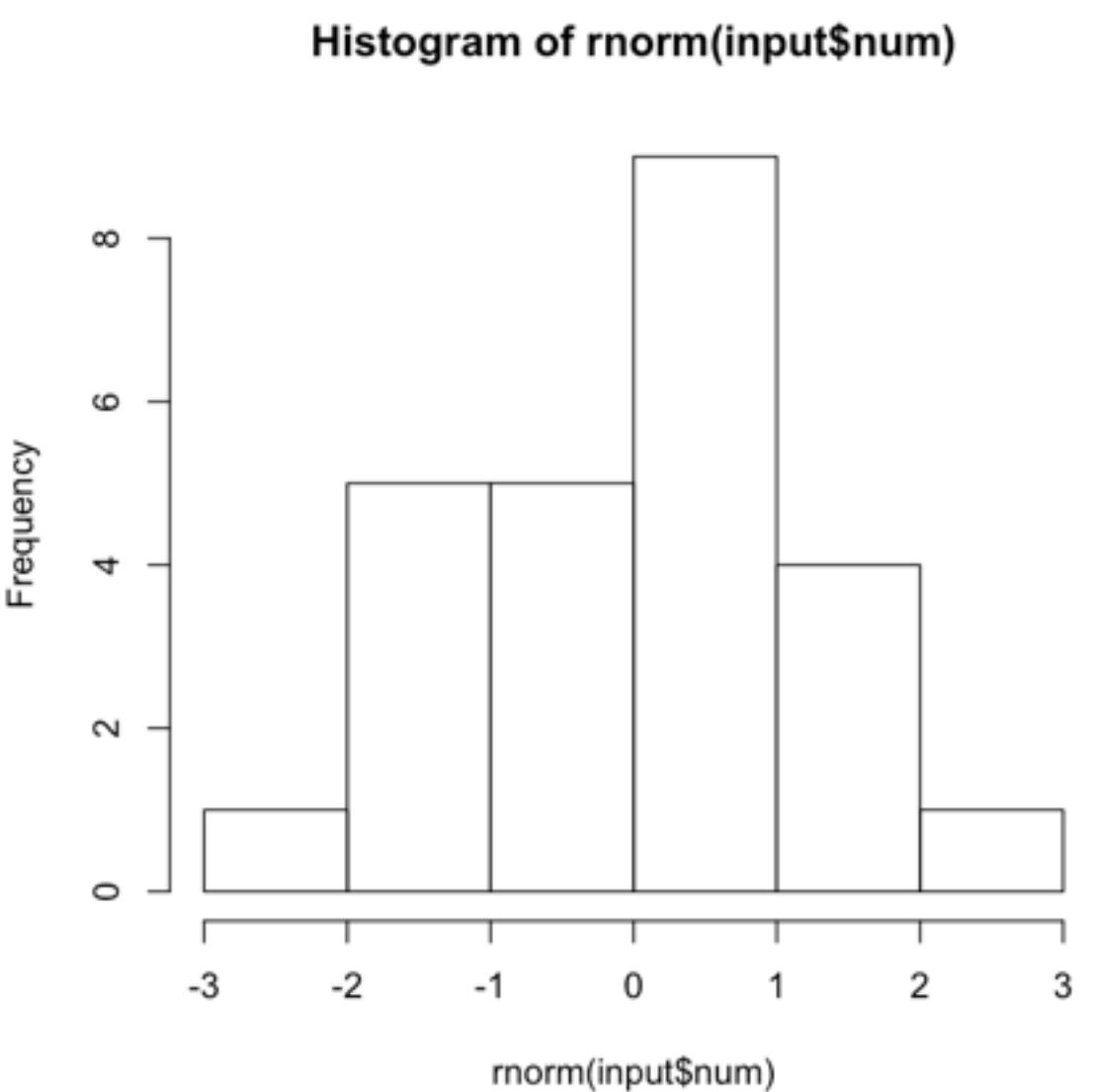
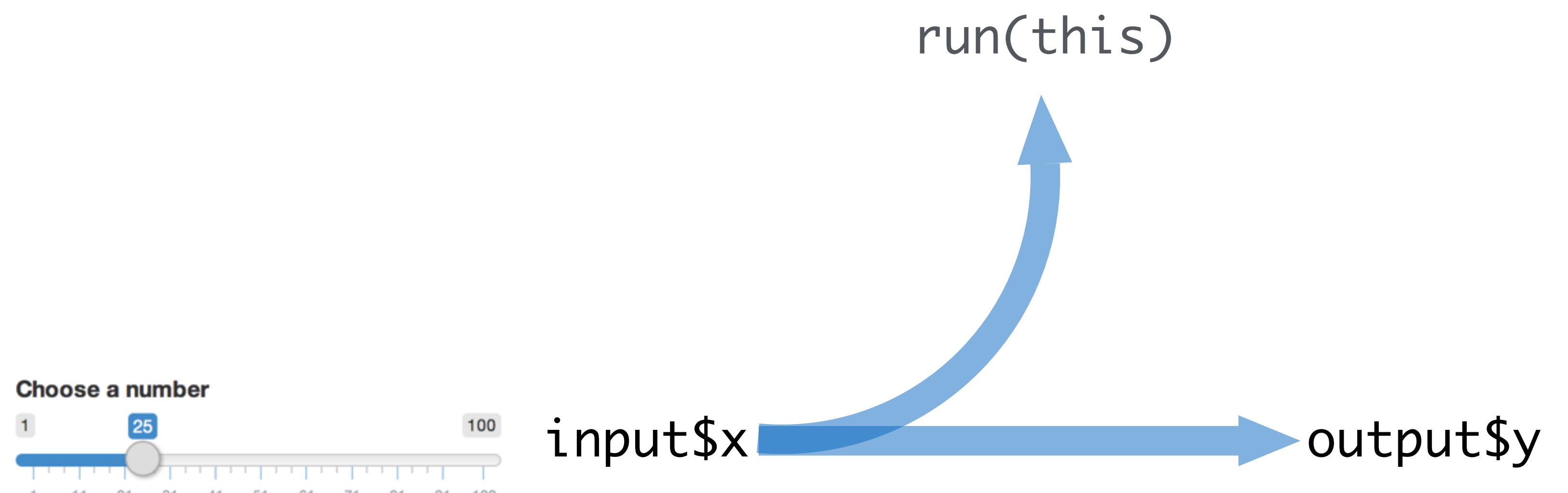
The screenshot shows a Microsoft Excel spreadsheet titled "Workbook1". The ribbon menu is visible with tabs for Tables, Charts, SmartArt, Formulas, Data, and Review. The "Formulas" tab is selected. The formula bar shows the formula `=1000` with a dropdown arrow. The main area displays a single cell with the value "1000". A large blue arrow points from this cell to the adjacent cell, which contains the value "1001". The ribbon also includes sections for Font, Alignment, Number, Format, and Cells, with various tools like Conditional Formatting, Styles, Insert, and Delete. The column headers C through O are visible at the top.

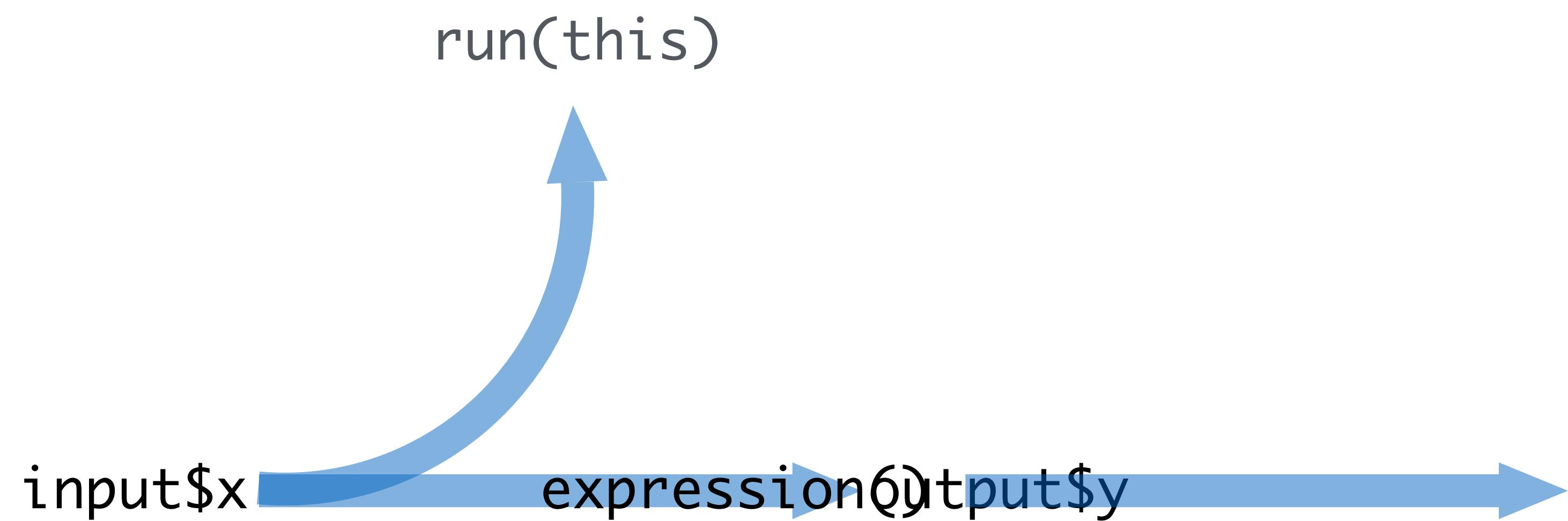
Workbook1

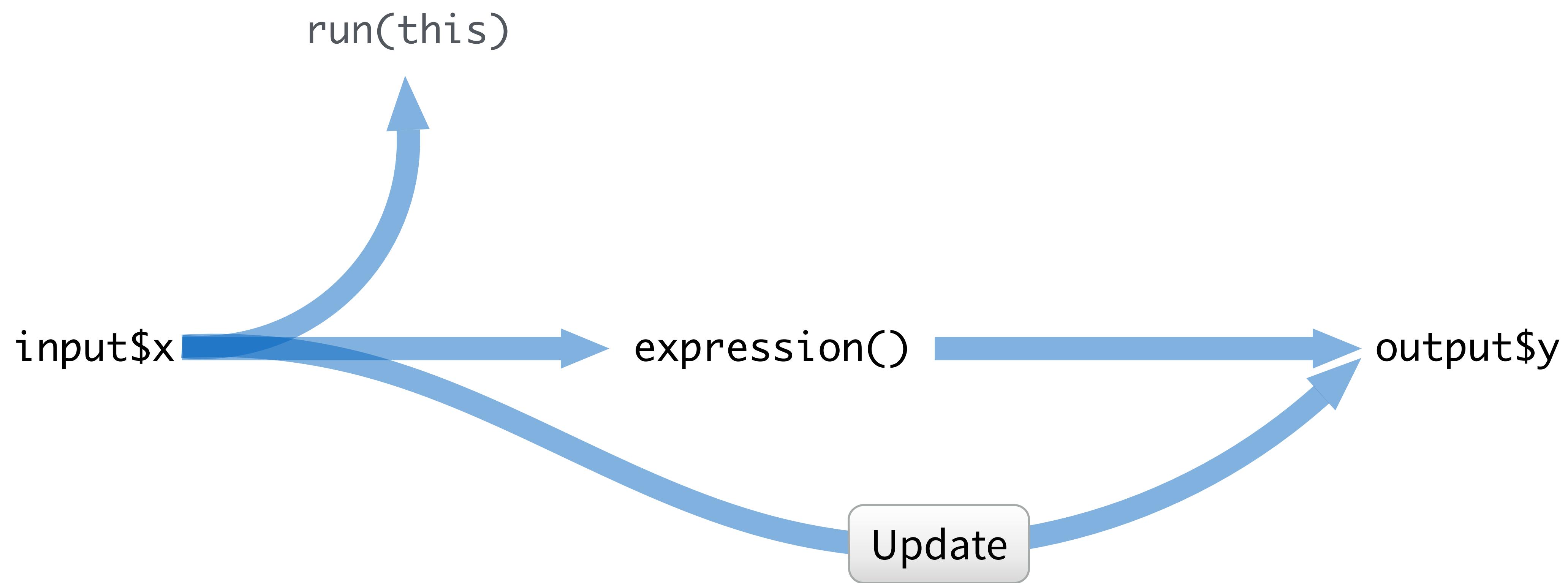
Tables Charts SmartArt Formulas Data Review

Font Alignment Number Format Cells

1000 → 1001
input\$x → output\$y



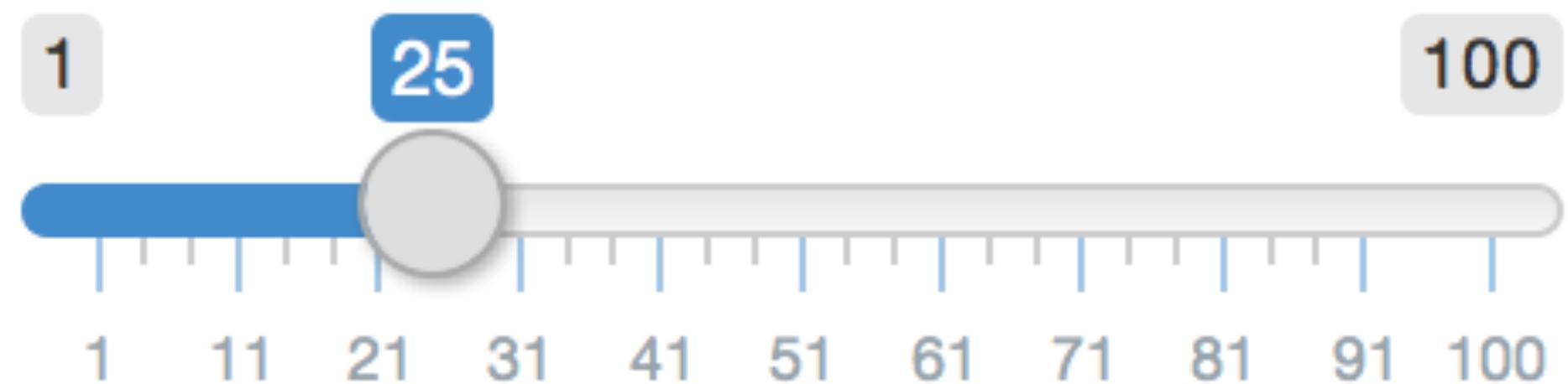




**Begin with
reactive values**

Syntax

Choose a number



```
sliderInput(inputId = "num", label = "Choose a number", ...)
```

this input will provide a value
saved as `input$num`

Input values

The input value changes whenever a user changes the input.

Choose a number

A slider input with a title "Choose a number". The slider has a blue track and a grey handle. The value "25" is displayed in a blue box above the slider. The slider scale shows tick marks at 1, 11, 21, 31, 41, 51, 61, 71, 81, 91, and 100.

input\$num = 25

Choose a number

A slider input with a title "Choose a number". The slider has a blue track and a grey handle. The value "50" is displayed in a blue box above the slider. The slider scale shows tick marks at 1, 11, 21, 31, 41, 51, 61, 71, 81, 91, and 100.

input\$num = 50

Choose a number

A slider input with a title "Choose a number". The slider has a blue track and a grey handle. The value "75" is displayed in a blue box above the slider. The slider scale shows tick marks at 1, 11, 21, 31, 41, 51, 61, 71, 81, 91, and 100.

input\$num = 75

Reactive values work together with reactive functions.
You cannot call a reactive value from outside of one.



```
renderPlot({ hist(rnorm(100, input$num)) })
```



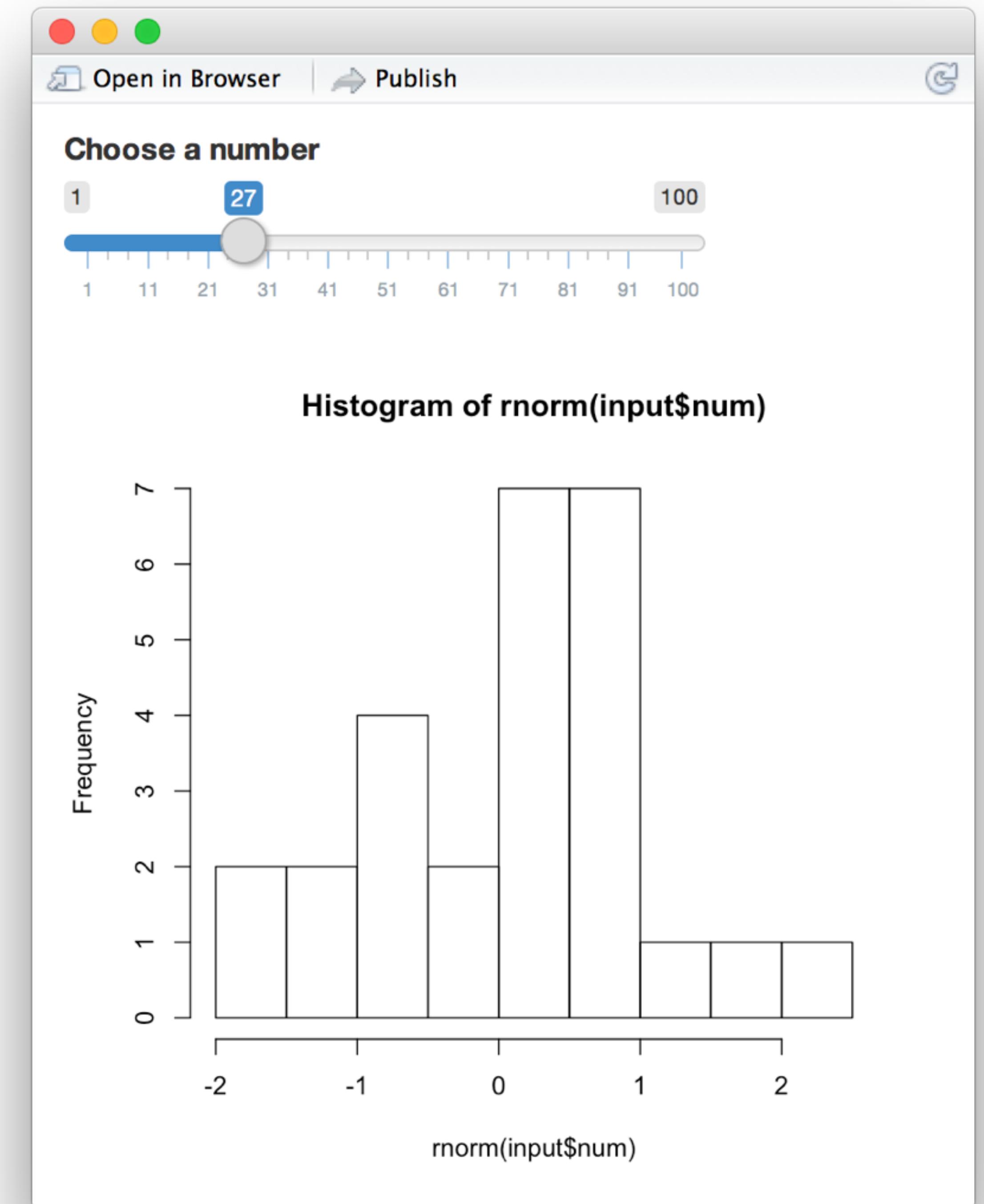
```
hist(rnorm(100, input$num))
```

```
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)

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

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

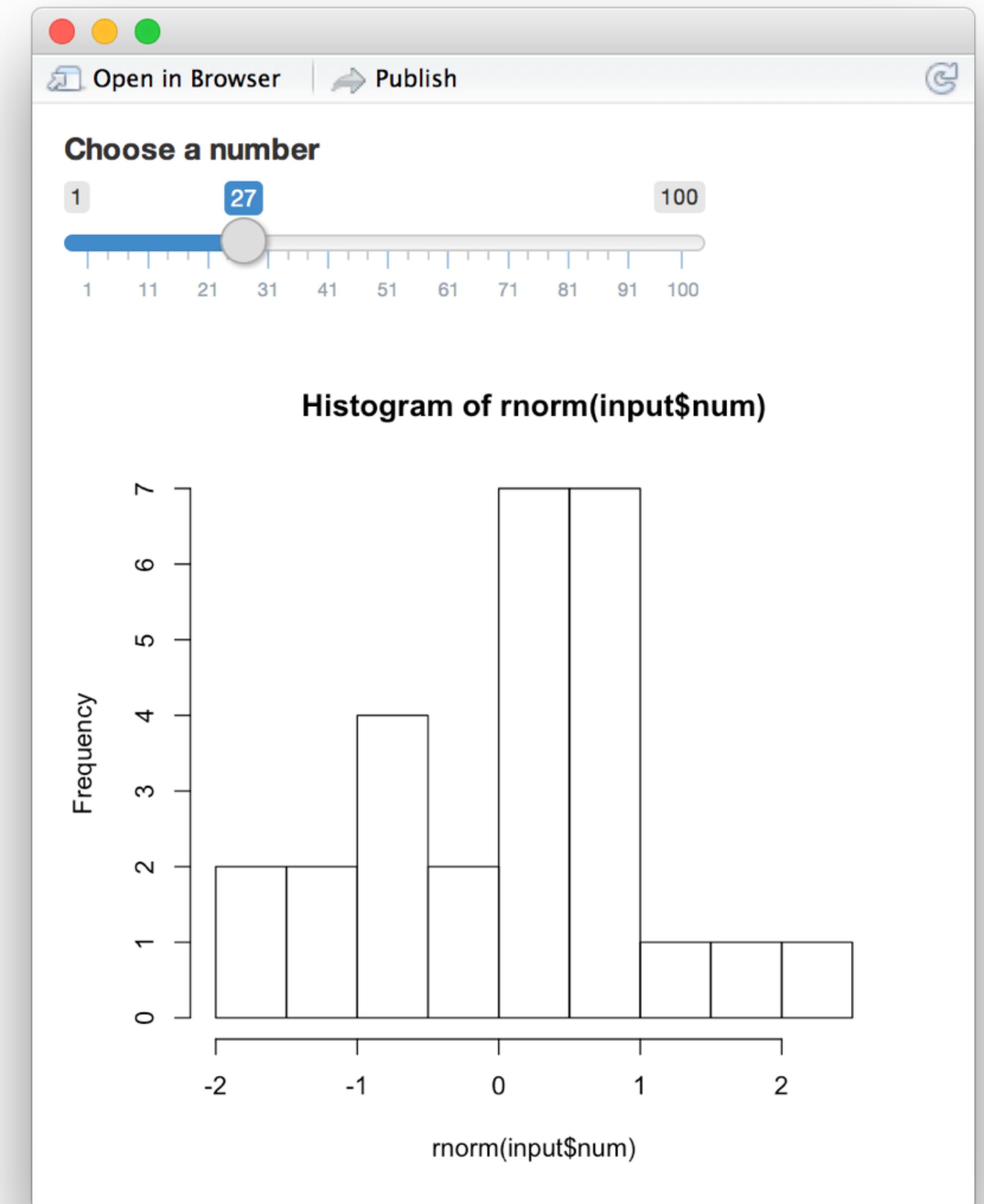


```
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)

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

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



```
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)

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

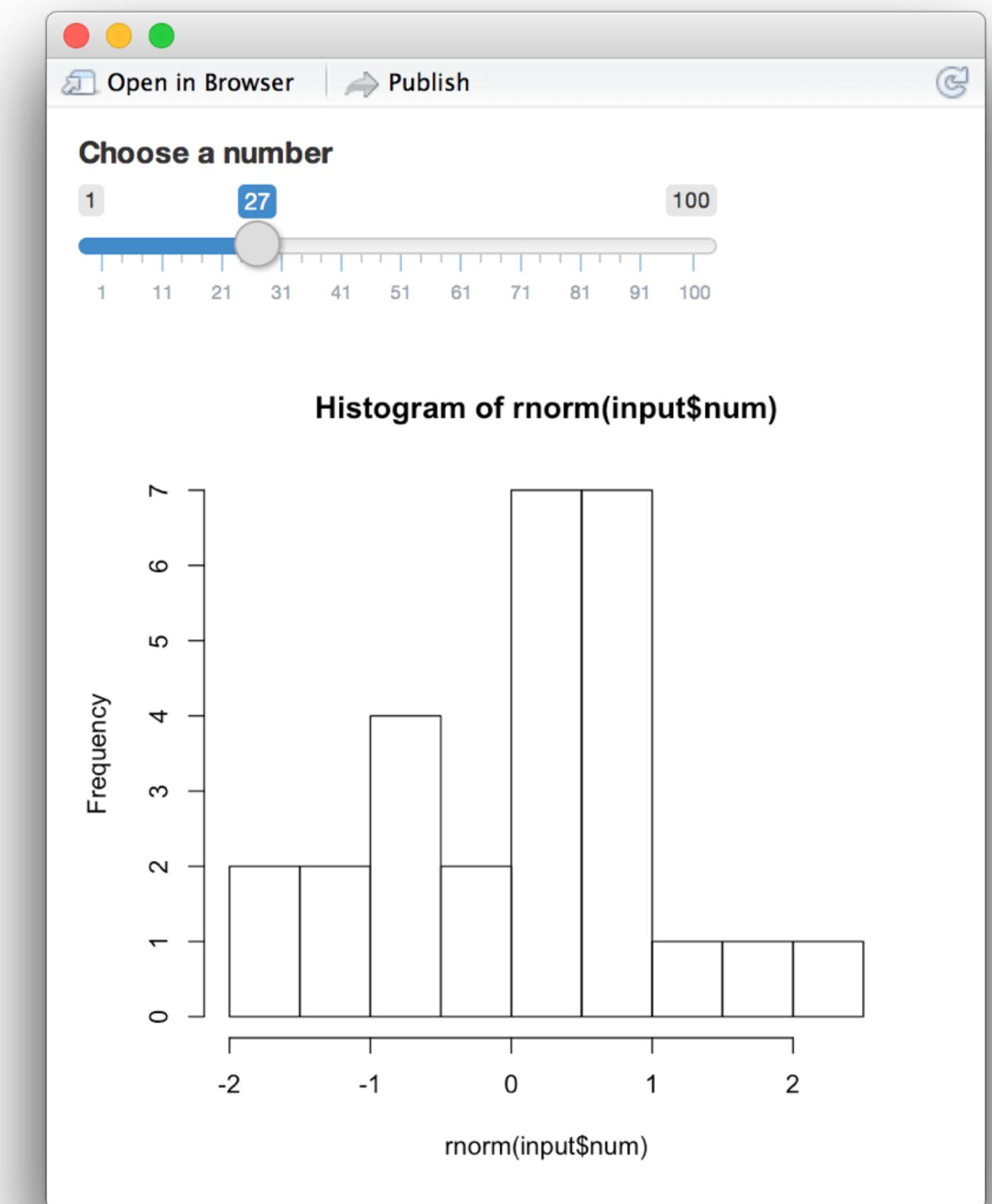
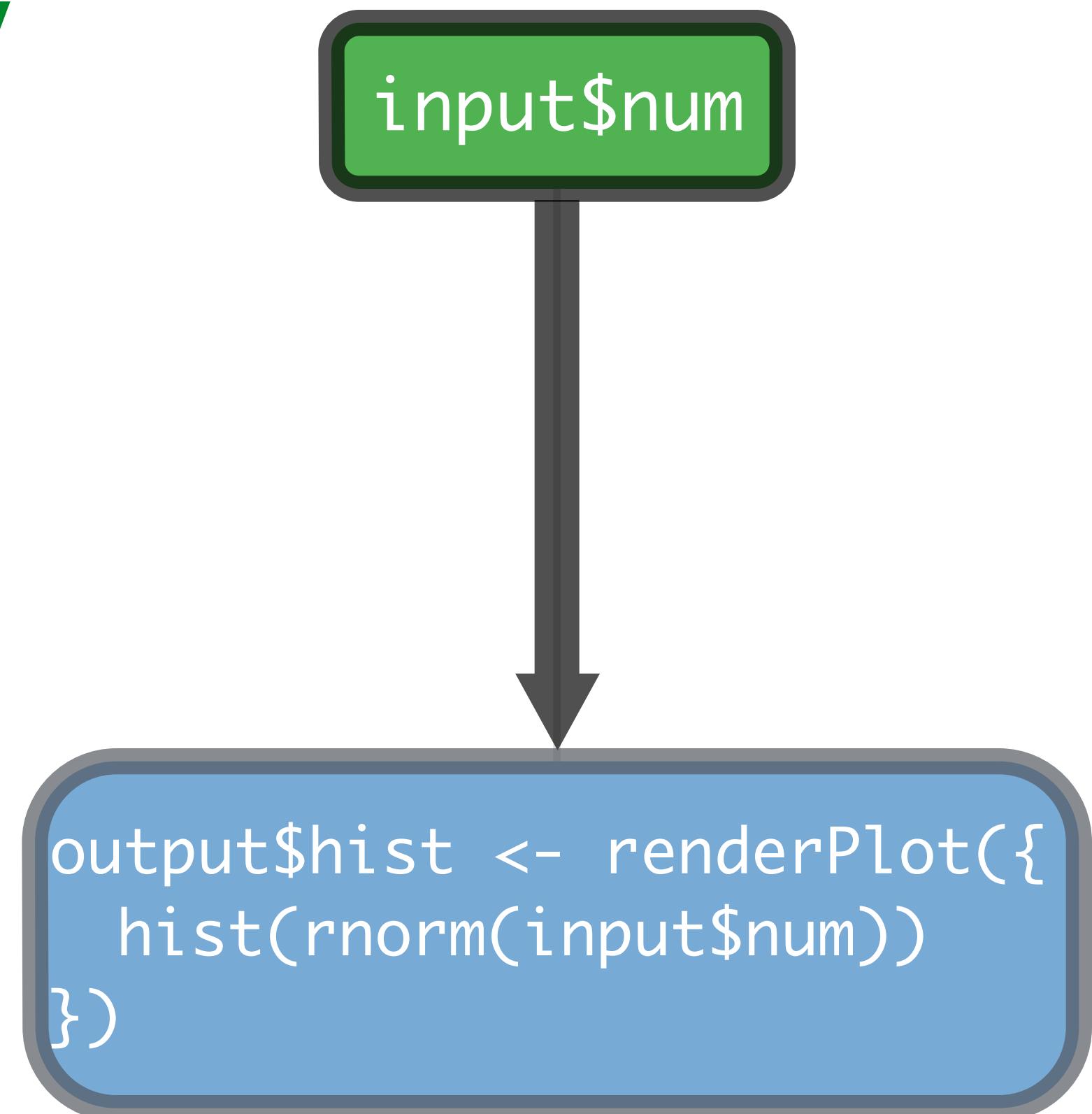
shinyApp(ui = ui, server = server)
```

Error in .getReactiveEnvironment()
\$currentContext() :
Operation not allowed without an
active reactive context. (You tried
to do something that can only be done
from inside a reactive expression or
observer.)

Think of reactivity in R as a two step process

1 Reactive values notify

the functions that use them
when they become invalid

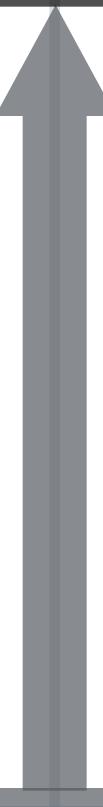


Think of reactivity in R as a two step process

1 Reactive values notify

the functions that use them
when they become invalid

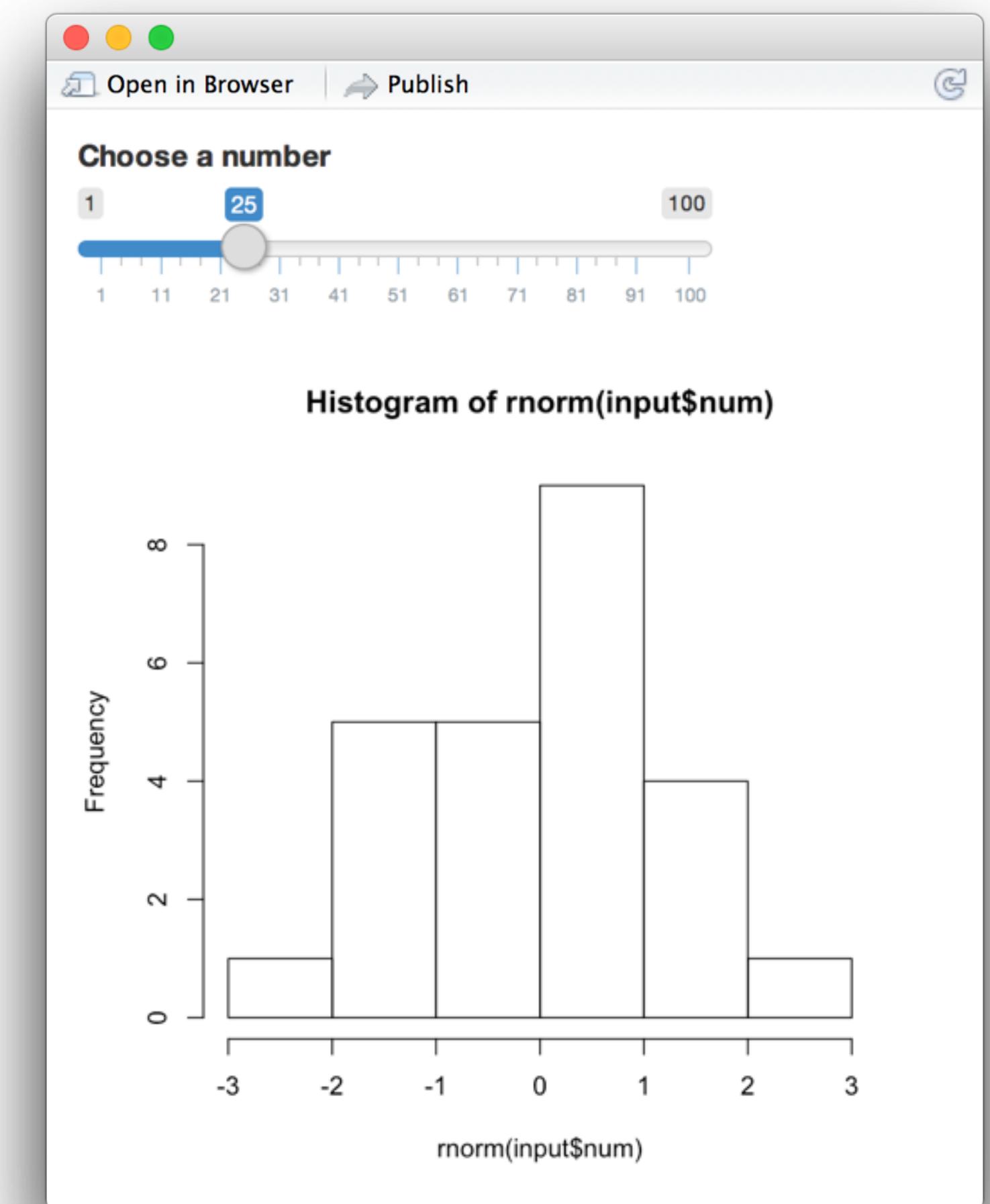
input\$num



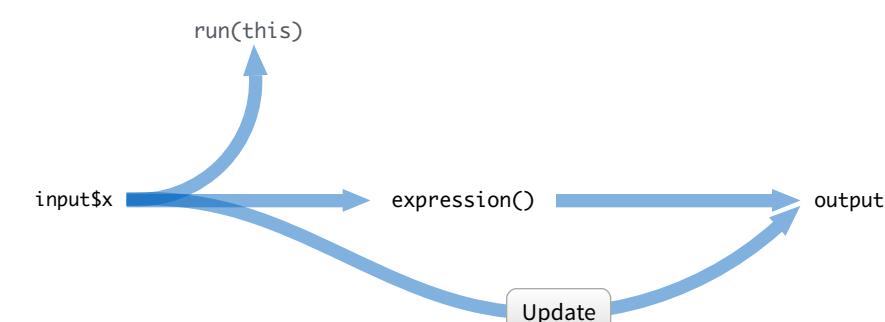
2 The objects created by reactive functions respond

(different objects respond differently)

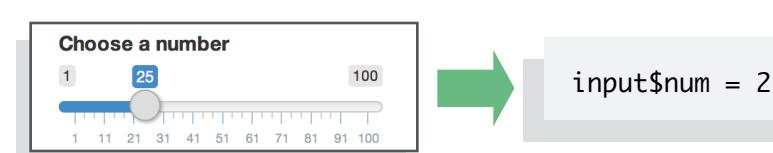
```
output$hist <- renderPlot({  
  hist(rnorm(input$num))  
})
```



Recap: Reactive values



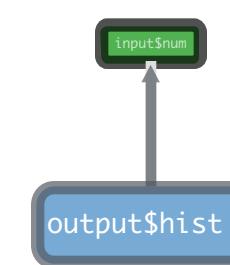
Reactive values act as the data streams that flow through your app.



The **input** list is a list of reactive values. The values show the current state of the inputs.



You can only call a reactive value from a function that is designed to work with one



Reactive values notify. The objects created by **reactive functions respond.**

Reactive toolkit

(7 indispensable functions)

Reactive functions

- 1** Use a code chunk to build (and rebuild) an object
 - **What code** will the function use?

- 2** The object will respond to changes in a set of reactive values
 - **Which reactive values** will the object respond to?

**Display output
with render*()**

Render functions build output to display in the app

function	creates
renderDataTable()	An interactive table <small>(from a data frame, matrix, or other table-like structure)</small>
renderImage()	An image (saved as a link to a source file)
renderPlot()	A plot
renderPrint()	A code block of printed output
renderTable()	A table <small>(from a data frame, matrix, or other table-like structure)</small>
renderText()	A character string
renderUI()	a Shiny UI element

render*()

Builds reactive output to display in UI

```
renderPlot( { hist(rnorm(input$num)) })
```

object will respond to *every reactive value in the code*

code used to build (and rebuild) object

render*()

Builds reactive output to display in UI

```
renderPlot( { hist(rnorm(input$num)) })
```

When notified that it is invalid, the object created by a render*() function **will rerun the entire block of code** associated with it

```

# 01-two-inputs

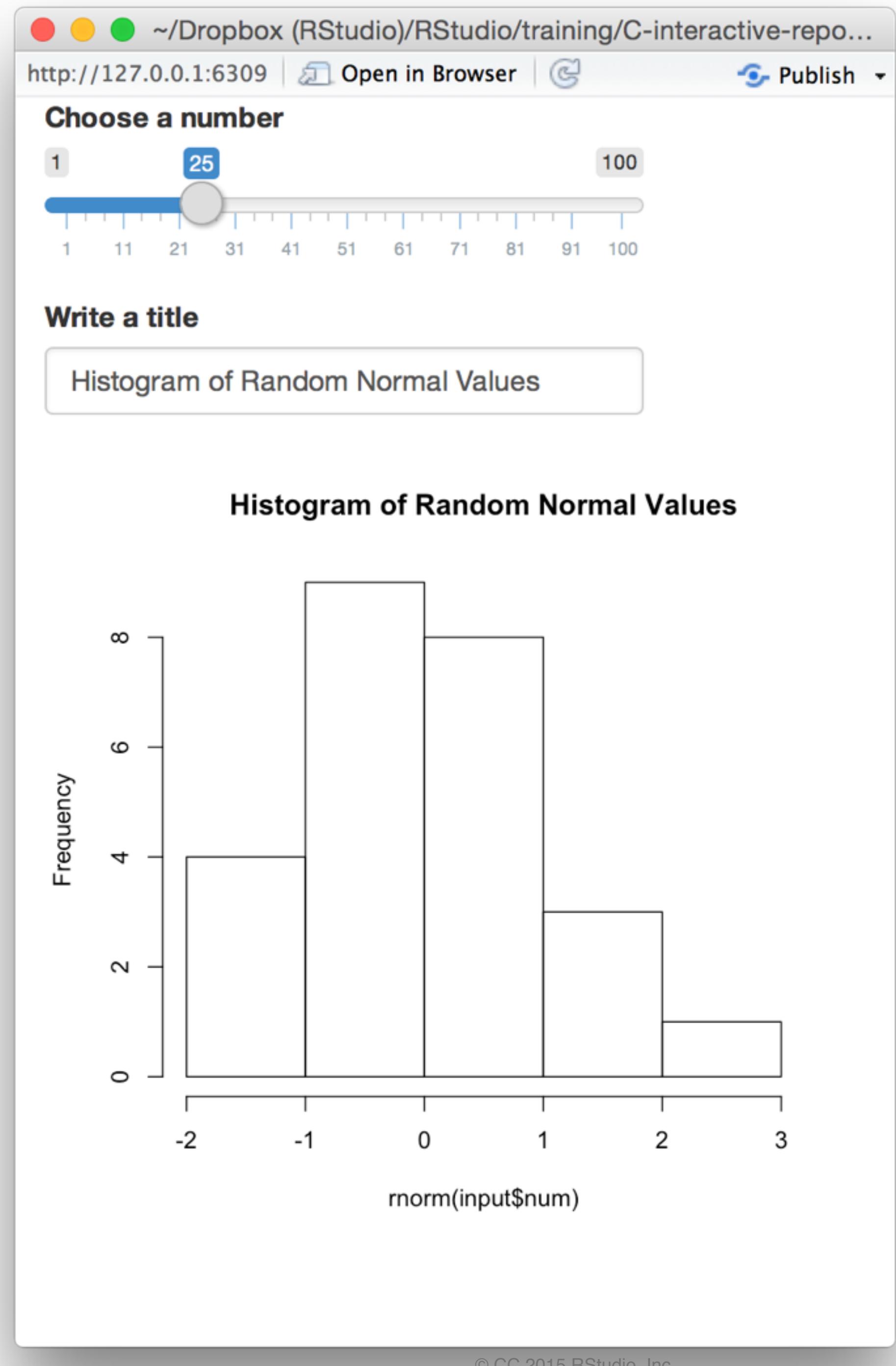
library(shiny)

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")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num), main = input$title)
  })
}

shinyApp(ui = ui, server = server)

```



```

# 01-two-inputs

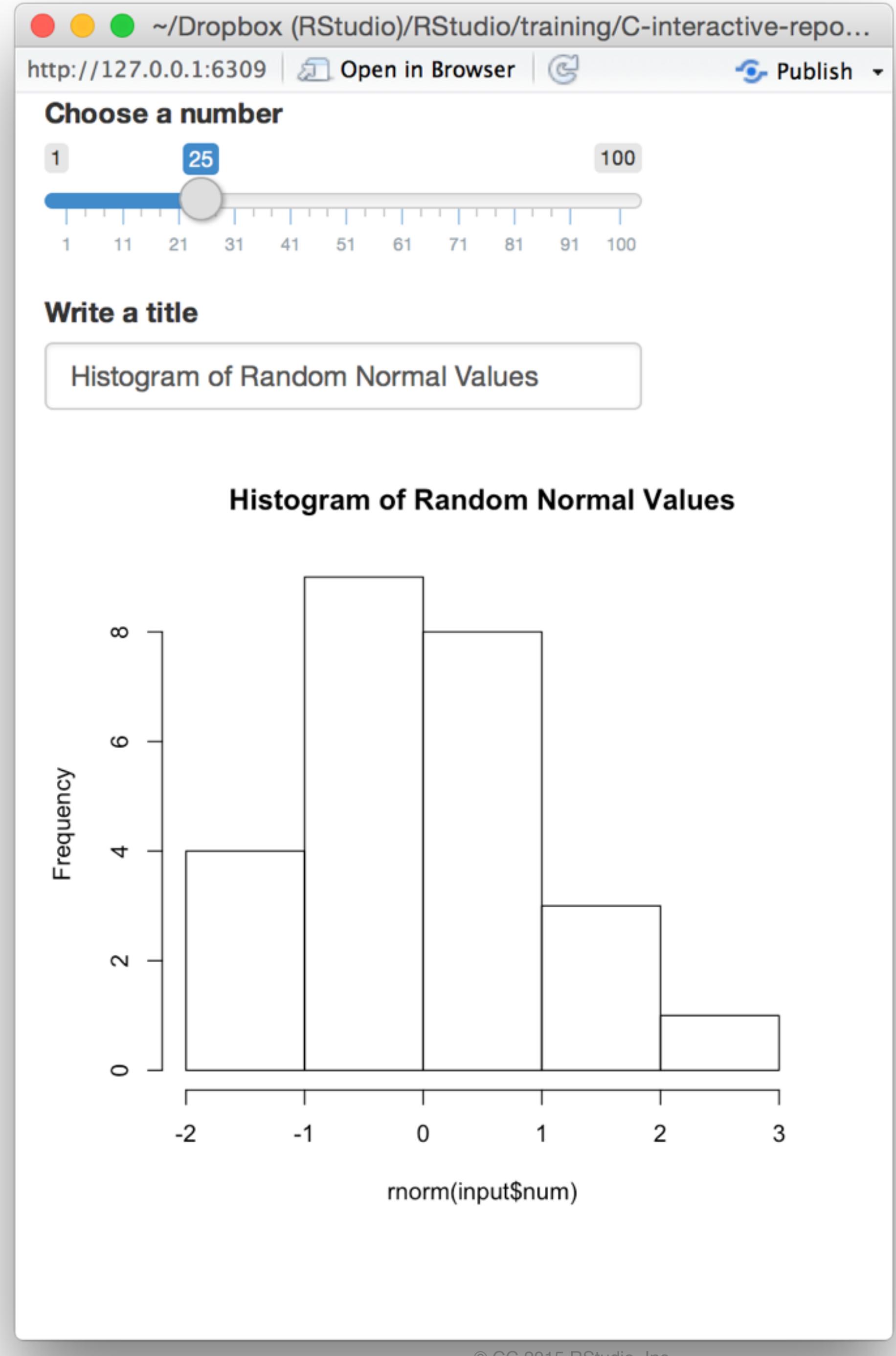
library(shiny)

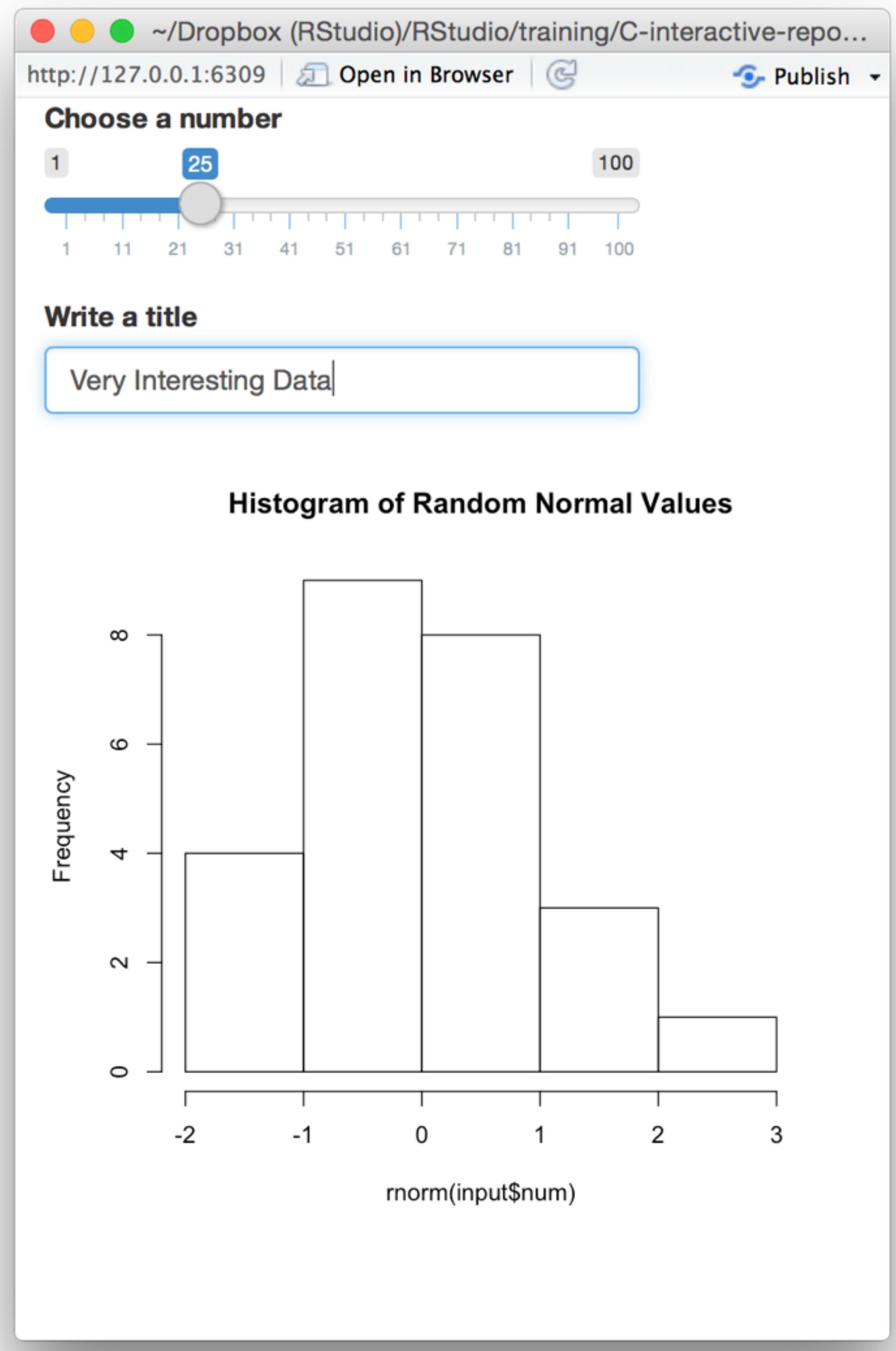
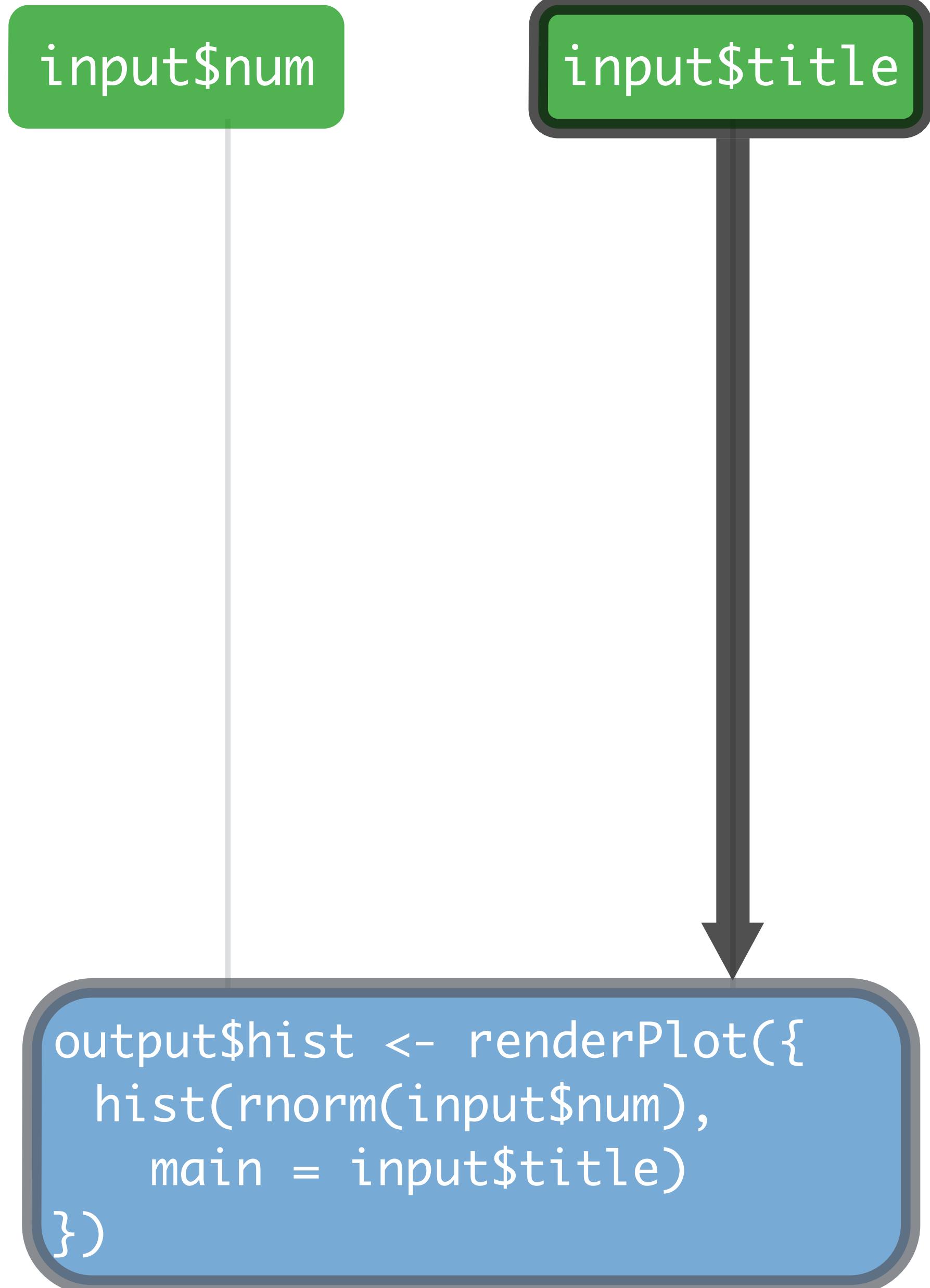
ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
 textInput(inputId = "title",
    label = "Write a title",
    value = "Histogram of Random Normal Values"),
  plotOutput("hist")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num), main = input$title)
  })
}

shinyApp(ui = ui, server = server)

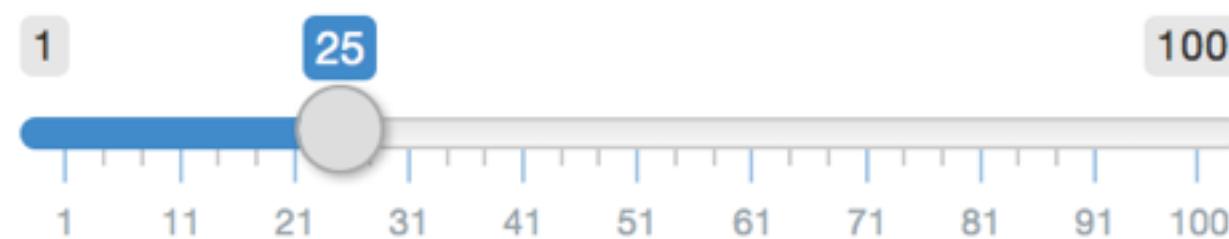
```





Choose a number

1 25 100

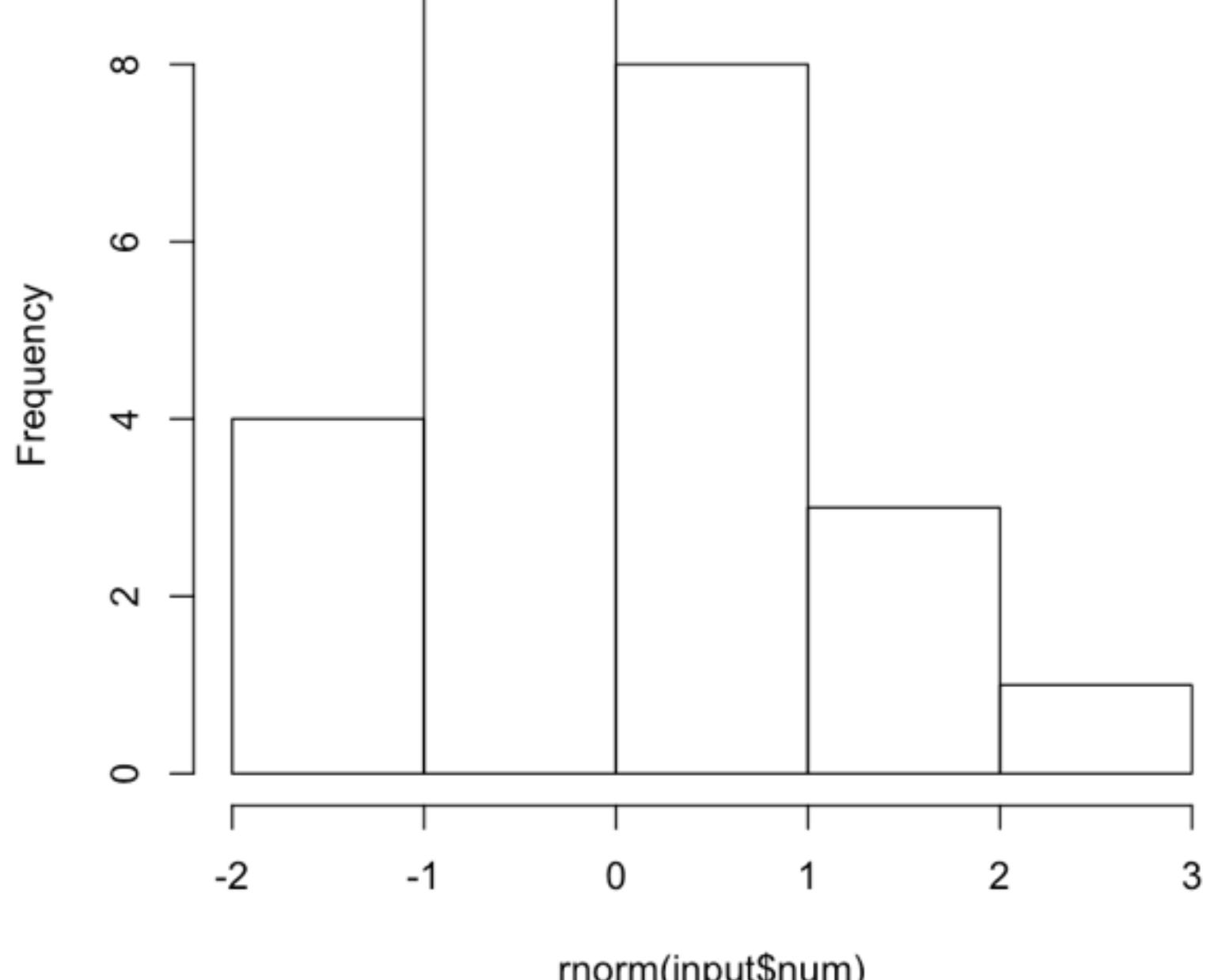


1 11 21 31 41 51 61 71 81 91 100

Write a title

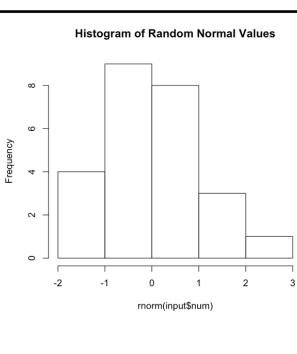
Very Interesting Data

Histogram of Random Normal Values



```
output$hist <- renderPlot({  
  hist(rnorm(input$num),  
        main = input$title)  
})
```

Recap: render*



render*() functions make **objects to display**

output\$

Always save the result to **output\$**

```
output$hist <- renderPlot({  
  hist(rnorm(input$num),  
    main = input$title)  
})
```

render*() makes an observer object that has a
block of code associated with it

```
renderPlot( { hist(rnorm(input$num)) })
```

The object will **rerun the entire code block**
to update itself whenever it is invalidated

Modularize code with reactive()

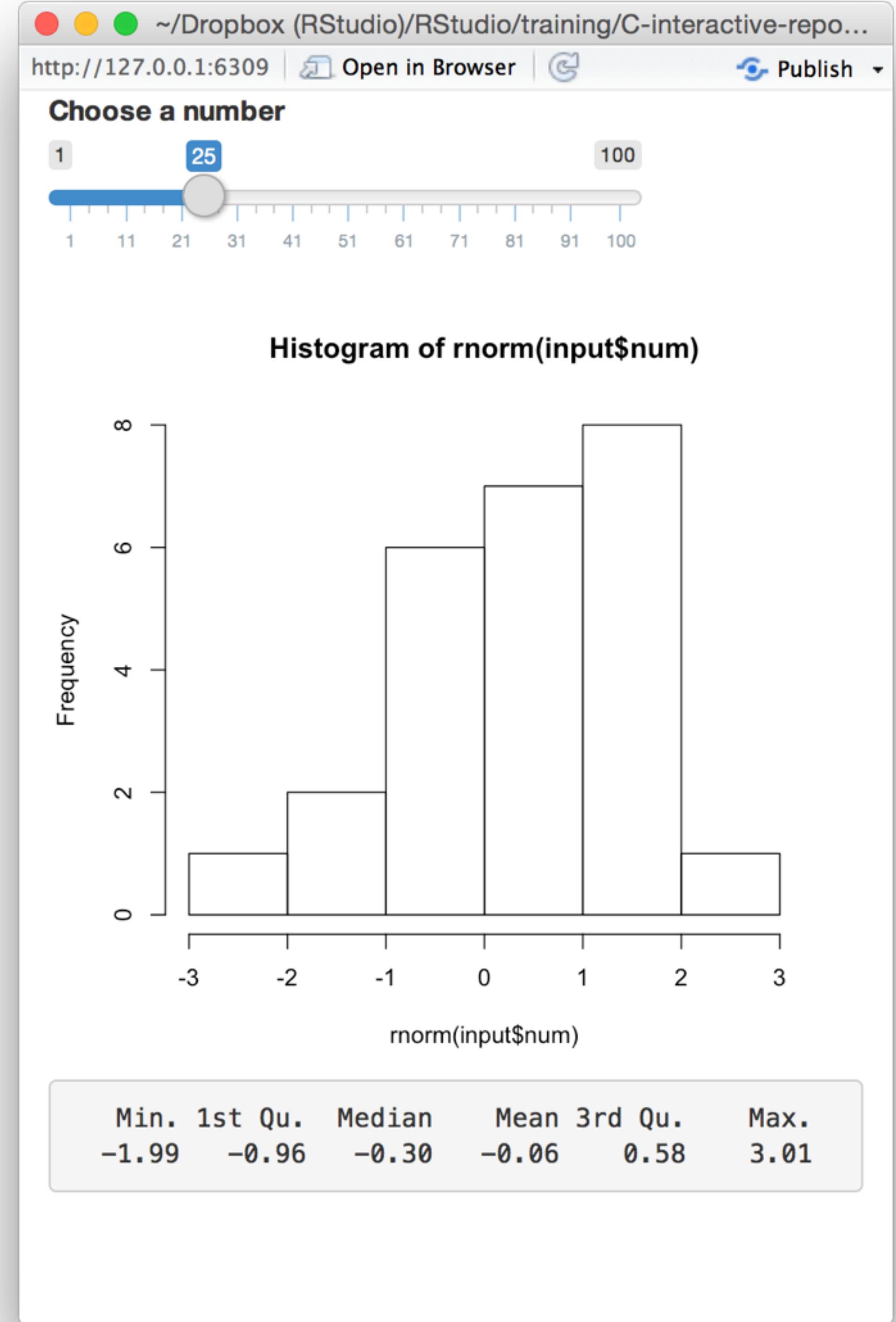
```
# 02-two-outputs

library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist"),
  verbatimTextOutput("stats")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
  output$stats <- renderPrint({
    summary(rnorm(input$num))
  })
}

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



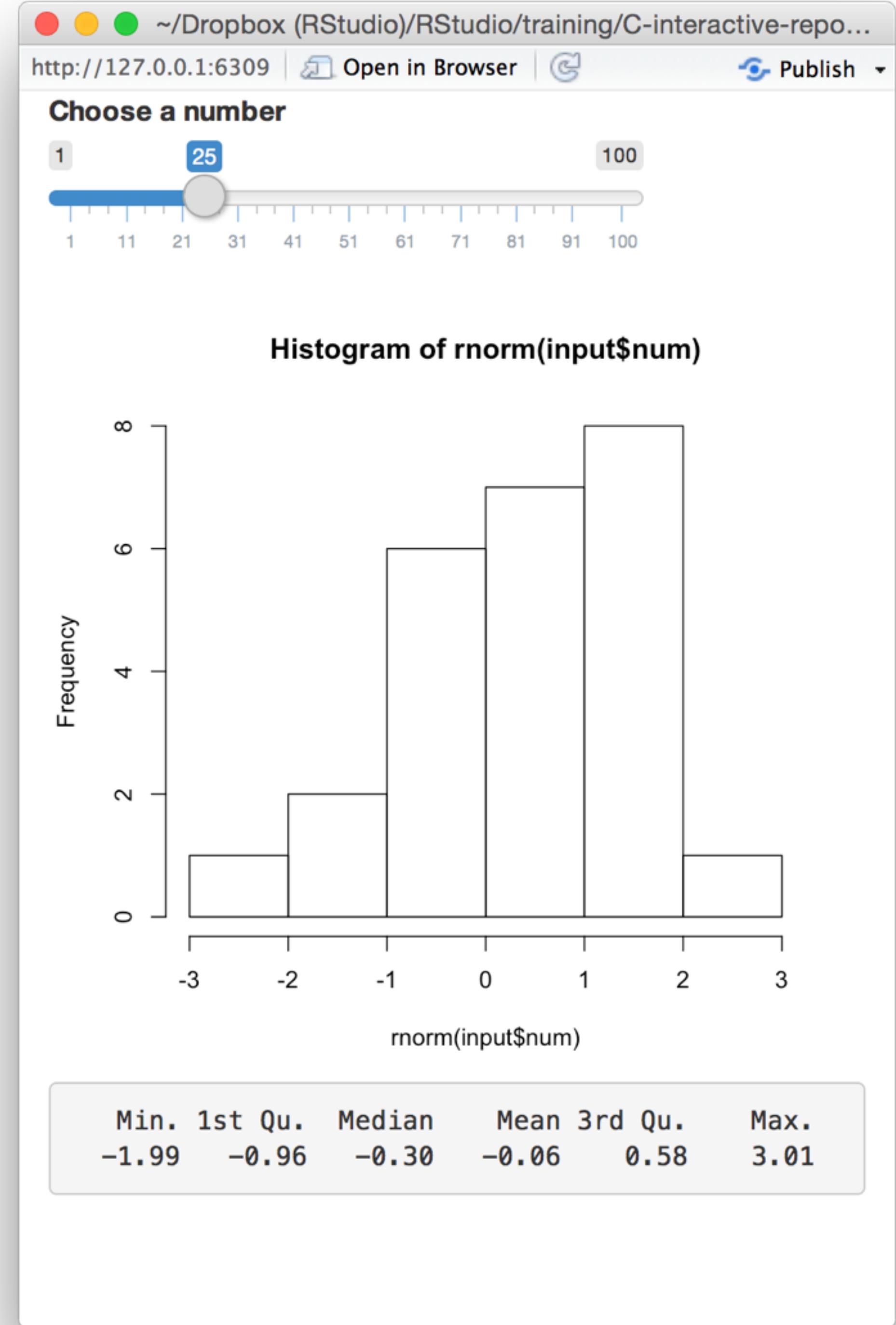
```
# 02-two-outputs

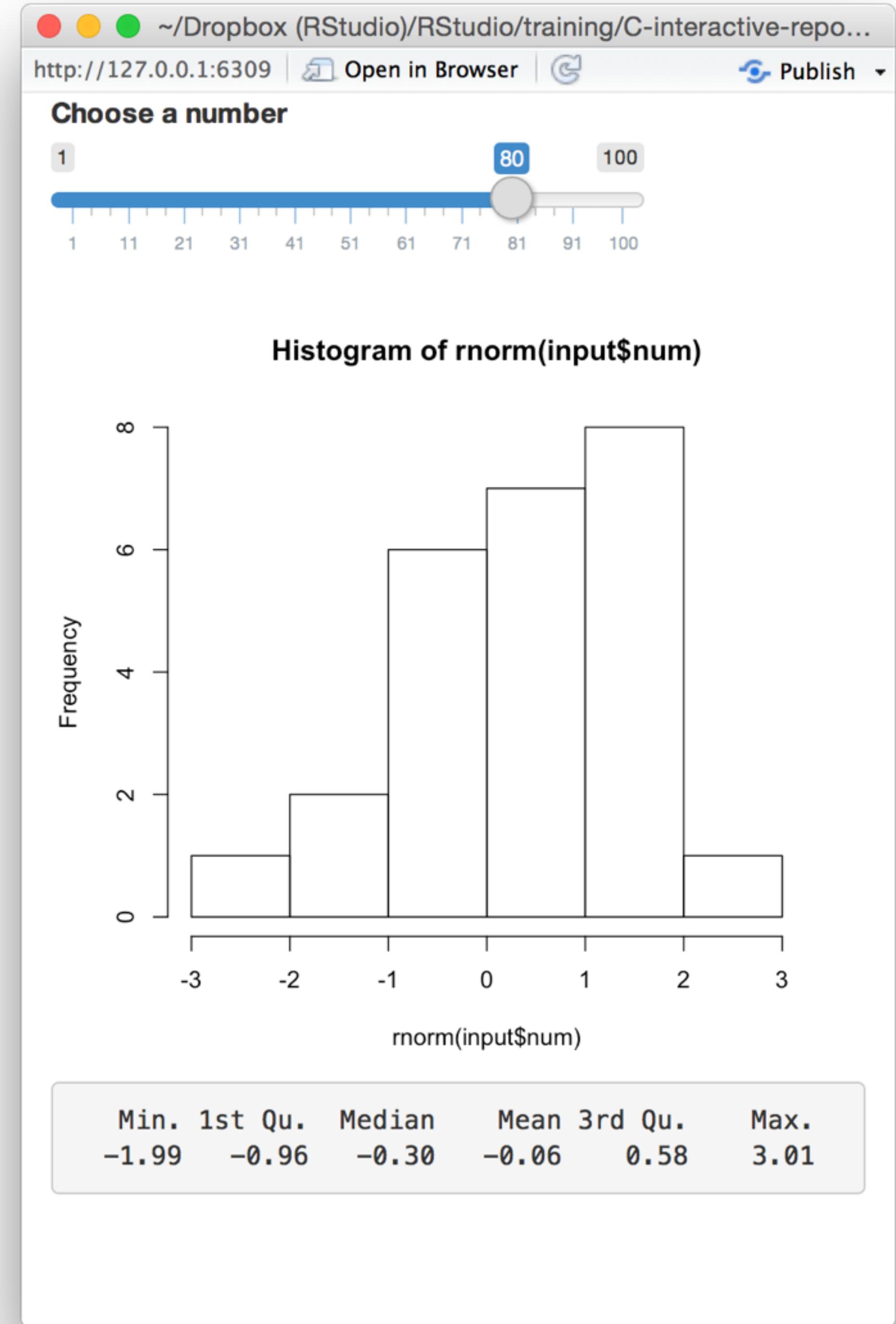
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist"),
  verbatimTextOutput("stats")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
  output$stats <- renderPrint({
    summary(rnorm(input$num))
  })
}

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

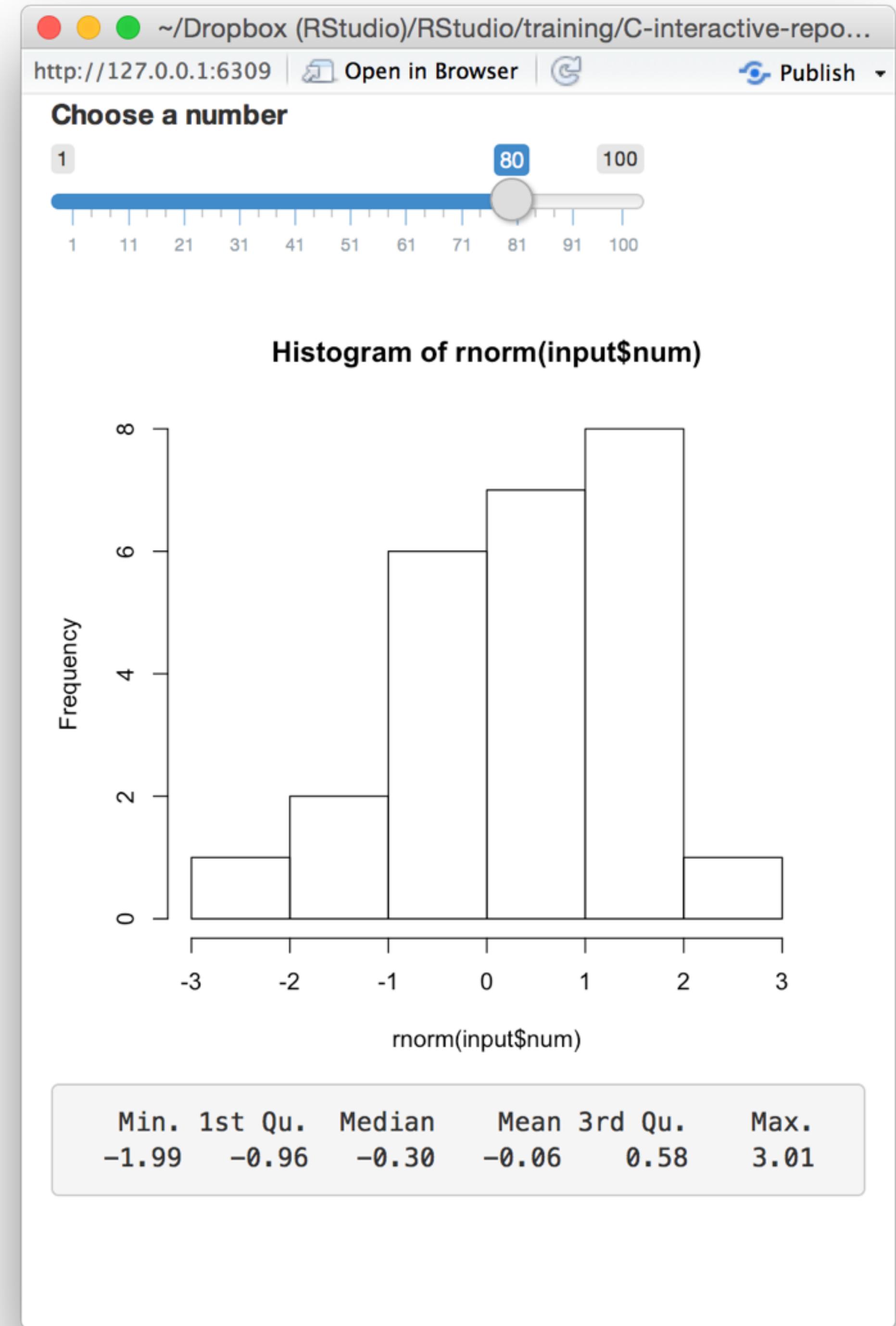




input\$num

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

```
output$stats <-  
  renderPrint({  
    summary(rnorm(input$num))  
  })
```



`input$num`

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

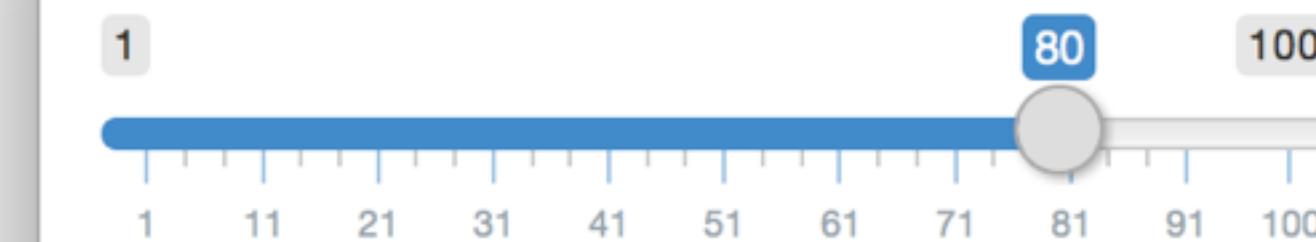
```
output$stats <-  
  renderPrint({  
    summary(rnorm(input$num))  
  })
```

input\$num

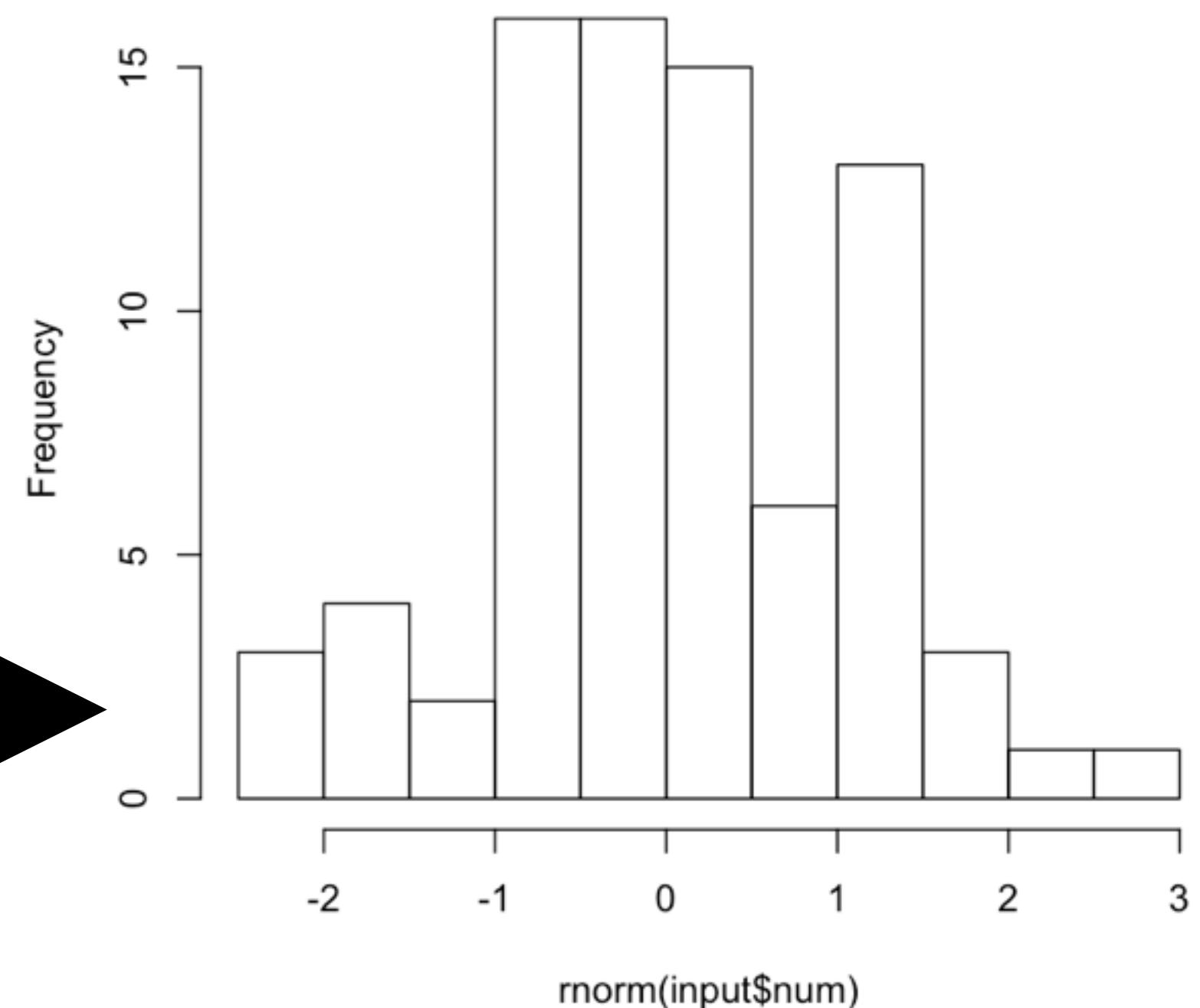
Can these describe
the same data?

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

```
output$stats <-  
  renderPrint({  
    summary(rnorm(input$num))  
  })
```



Histogram of `rnorm(input$num)`



Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-2.23	-0.66	0.11	0.11	0.72	2.14

input\$num

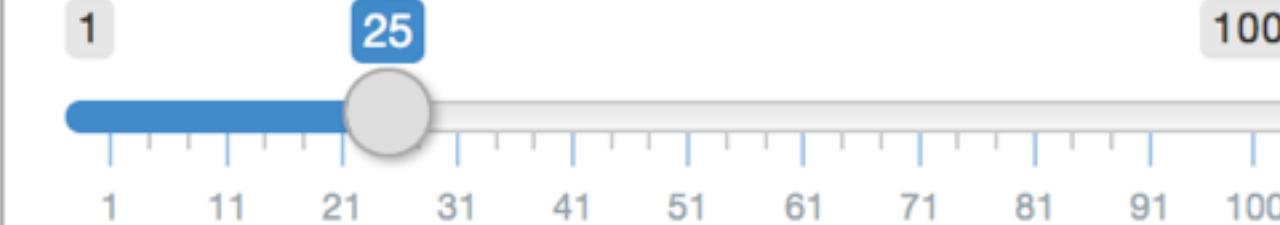
data <-? rnorm(input\$num)

```
output$hist <-  
  renderPlot({  
    hist(data)  
  })
```

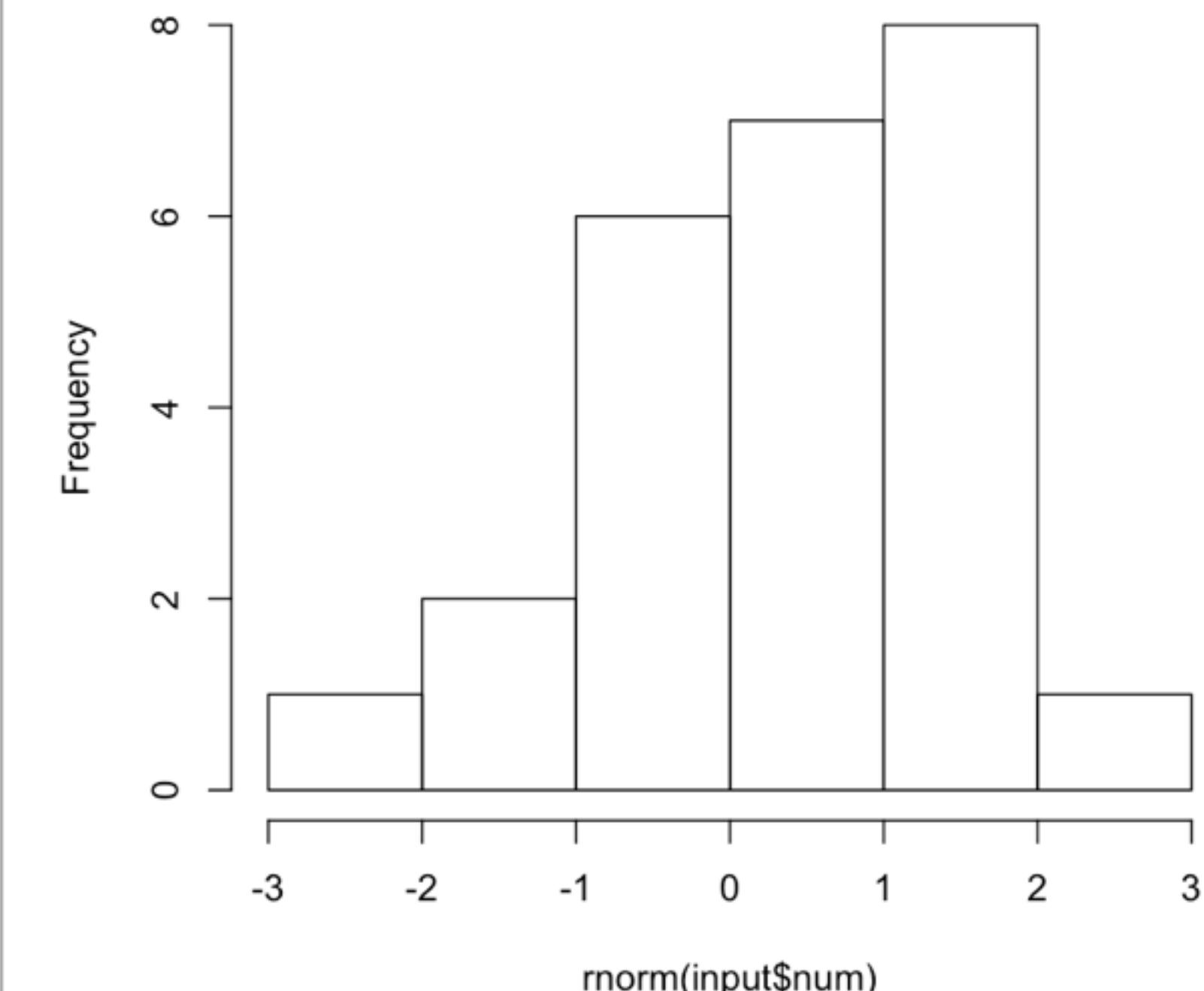
```
output$stats <-  
  renderPrint({  
    summary(data)  
  })
```

Choose a number

1 25 100



Histogram of rnorm(input\$num)



Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-1.99	-0.96	-0.30	-0.06	0.58	3.01

reactive()

Builds a reactive object (reactive expression)

```
data <- reactive( { rnorm(input$num) })
```

object will respond to *every reactive value in the code*

code used to build (and rebuild) object

A reactive expression is special in two ways

```
data()
```

- 1 You call a reactive expression like a function

```
# 02-two-outputs

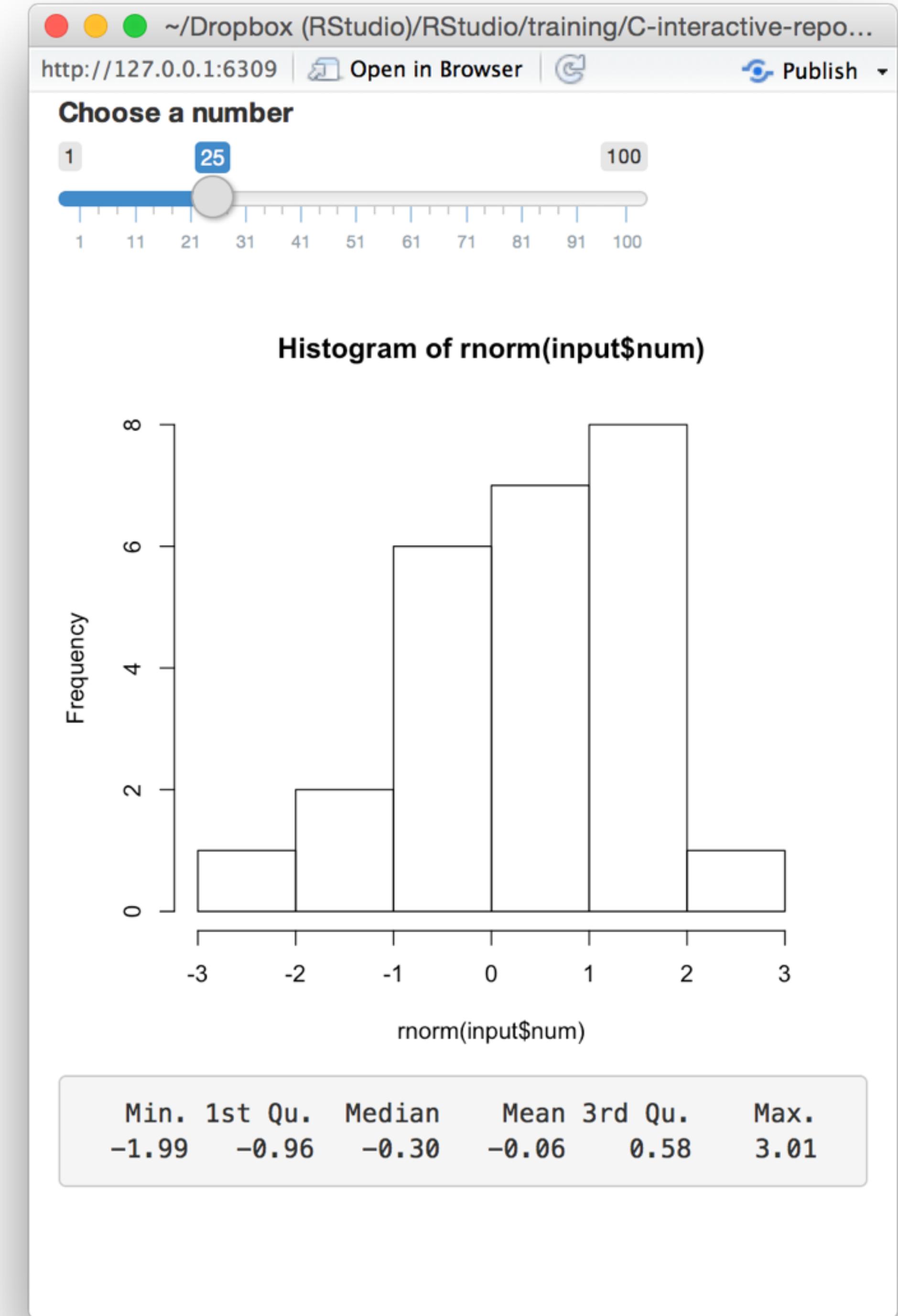
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist"),
  verbatimTextOutput("stats")
)

server <- function(input, output) {

  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
  output$stats <- renderPrint({
    summary(rnorm(input$num))
  })
}

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



```

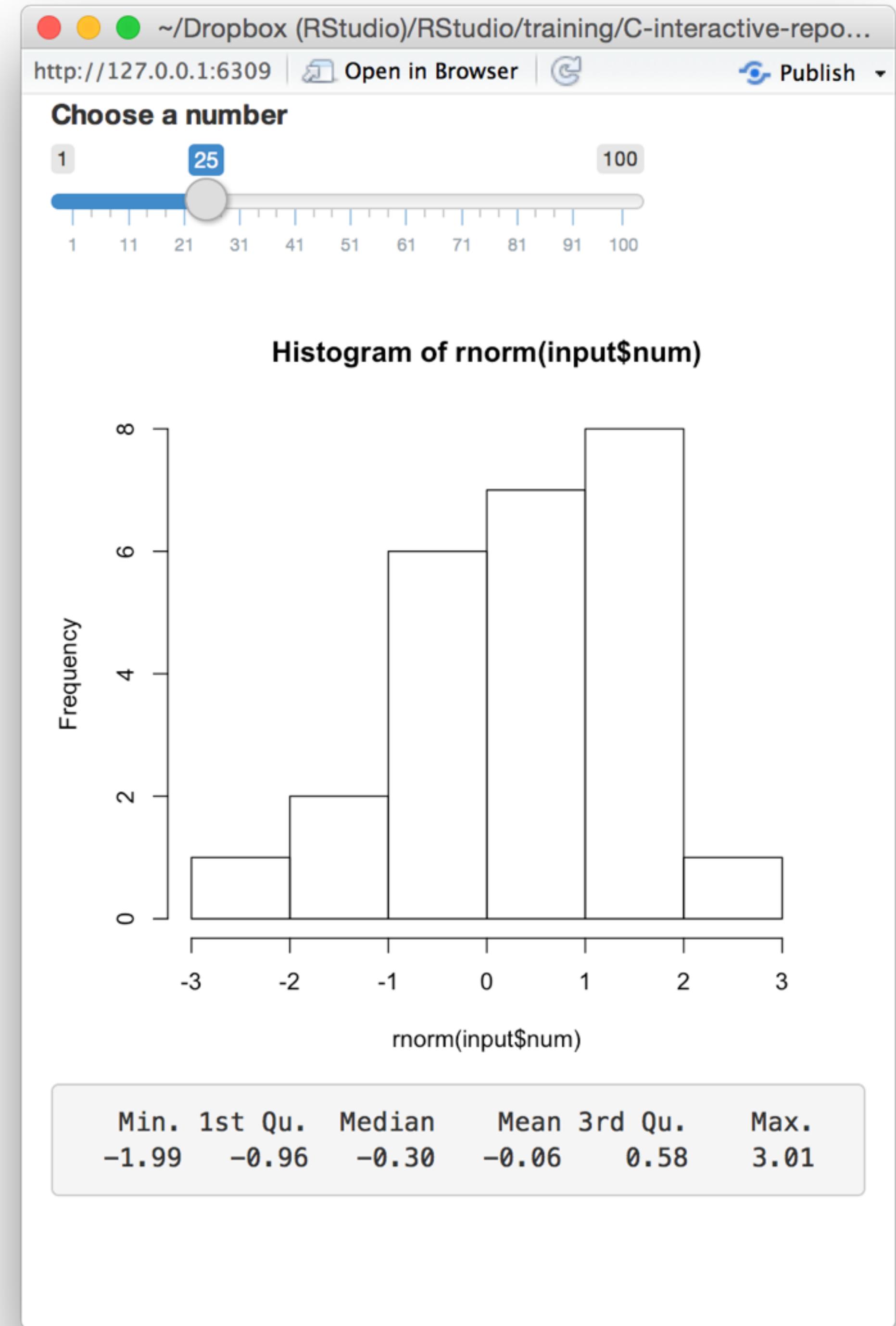
# 02-two-outputs

library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist"),
  verbatimTextOutput("stats")
)

server <- function(input, output) {
  data <- reactive({
    rnorm(input$num)
  })
  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
  output$stats <- renderPrint({
    summary(rnorm(input$num))
  })
}
shinyApp(ui = ui, server = server)

```

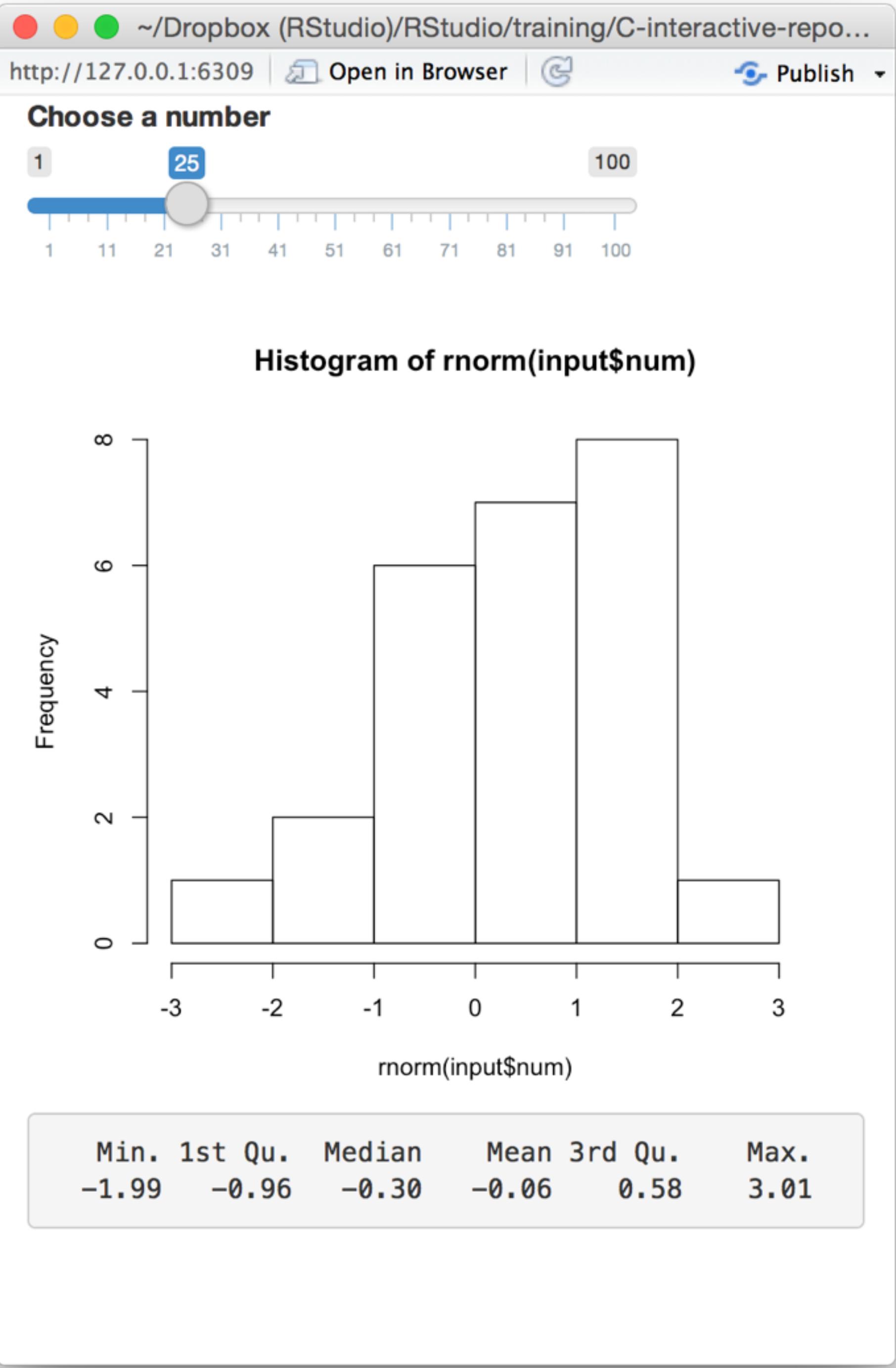


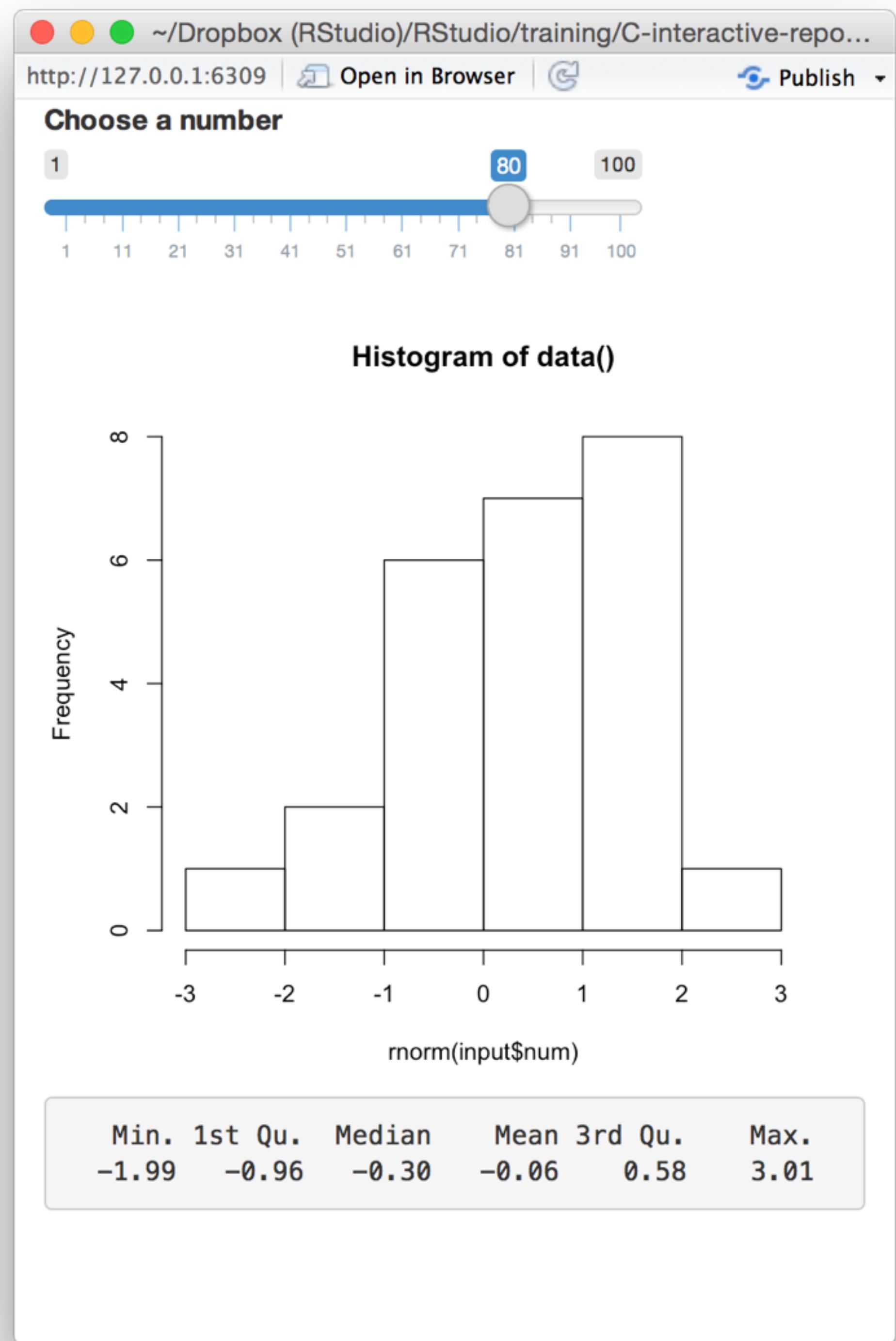
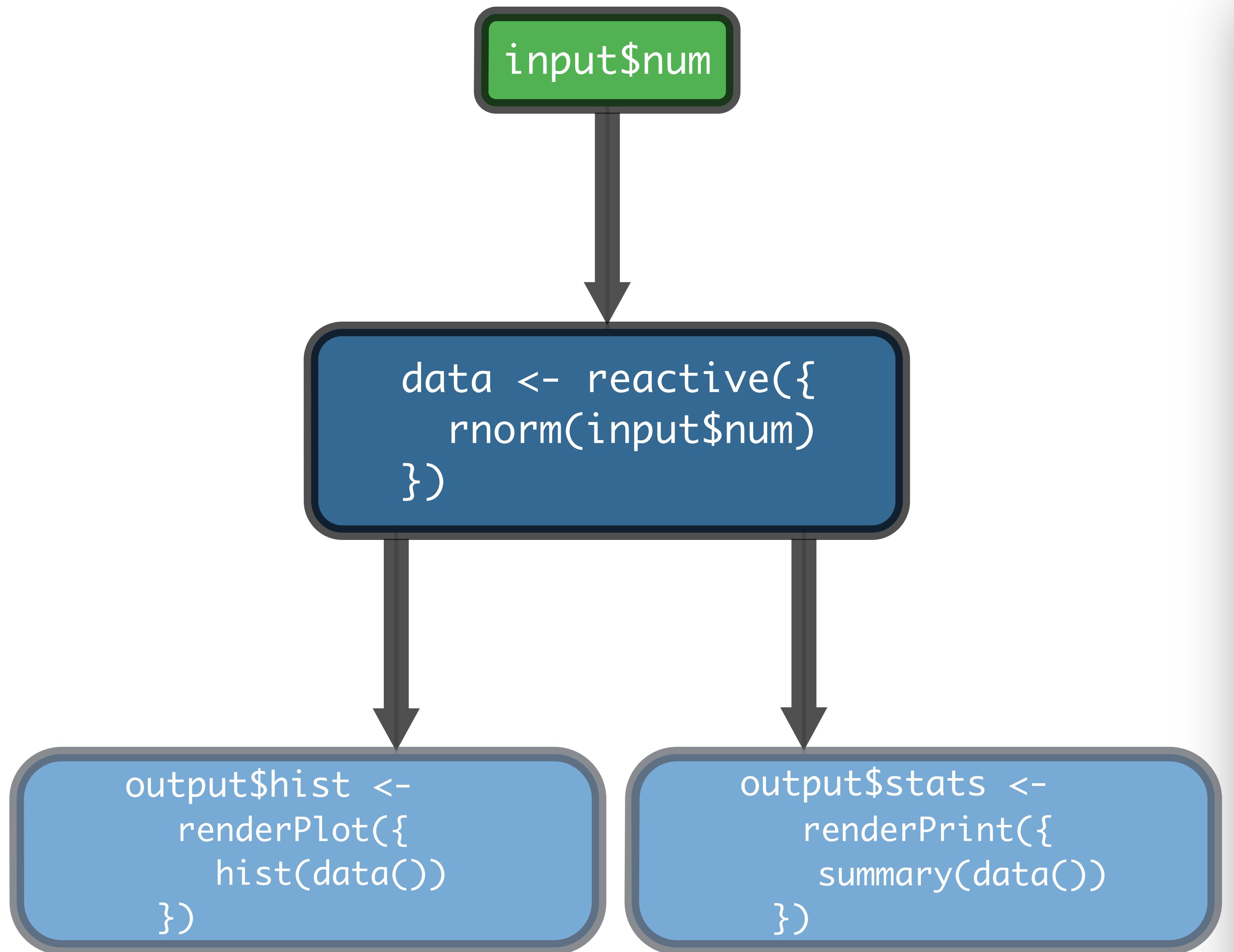
```
# 03-reactive

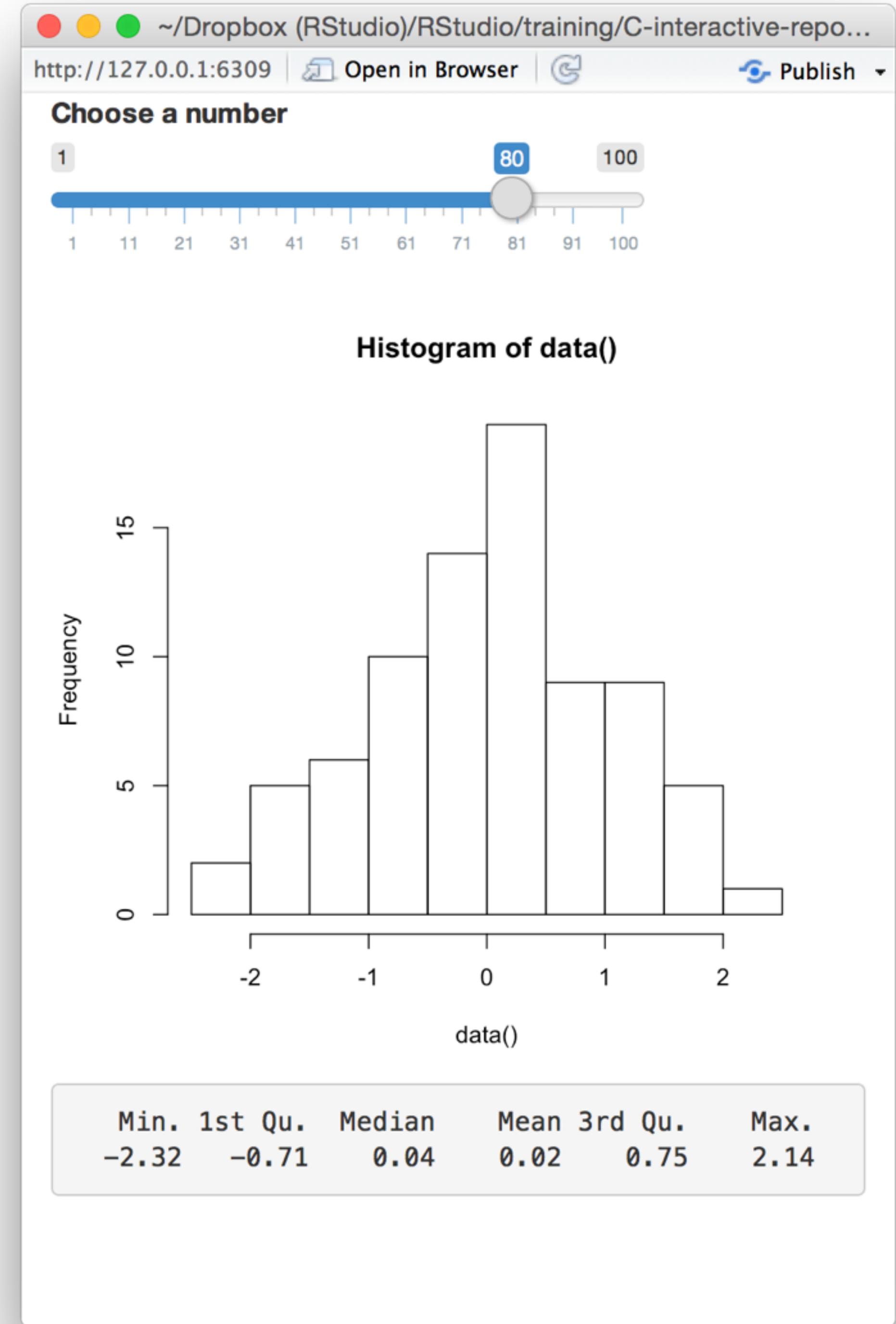
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist"),
  verbatimTextOutput("stats")
)

server <- function(input, output) {
  data <- reactive({
    rnorm(input$num)
  })
  output$hist <- renderPlot({
    hist(data())
  })
  output$stats <- renderPrint({
    summary(data())
  })
}
shinyApp(ui = ui, server = server)
```







input\$num

```
data <- reactive({  
  rnorm(input$num)  
})
```

```
output$hist <-  
  renderPlot({  
    hist(data())  
})
```

```
output$stats <-  
  renderPrint({  
    summary(data())  
})
```

A reactive expression is special in two ways

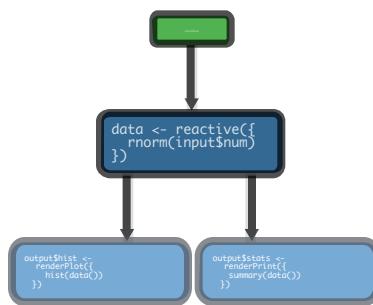
```
data()
```

- 1** You call a reactive expression like a function
- 2** Reactive expressions **cache** their values
(the expression will return its most recent value,
unless it has become invalidated)

Recap: reactive()

```
data <- reactive({  
  rnorm(input$num)  
})
```

reactive() makes an **object to use** (in downstream code)



Reactive expressions are themselves **reactive**. Use them to modularize your apps.

data()

Call a reactive expression like a **function**

2

Reactive expressions **cache** their values to avoid unnecessary computation

**Prevent reactions
with isolate()**

```
# 01-two-inputs

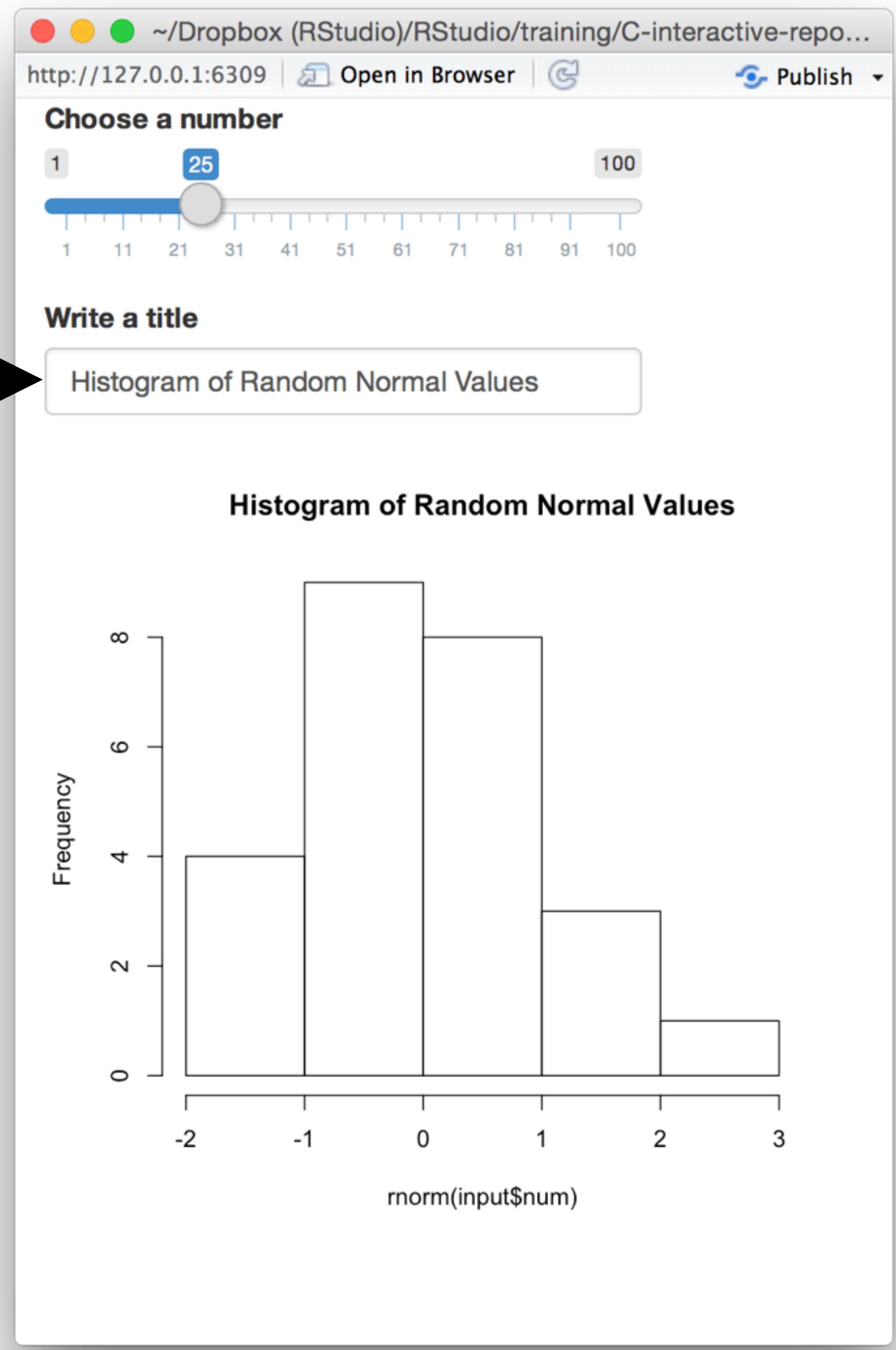
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  textFieldInput(inputId = "title",
    label = "Write a title",
    value = "Histogram of Random Normal Values"),
  plotOutput("hist")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num),
      main = input$title)
  })
}

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

**Can we prevent
the title field from
updating the plot?**



isolate()

Returns the result as a non-reactive value

```
isolate({ rnorm(input$num) })
```

object will NOT respond to
*any reactive value in the
code*

code used to build
object

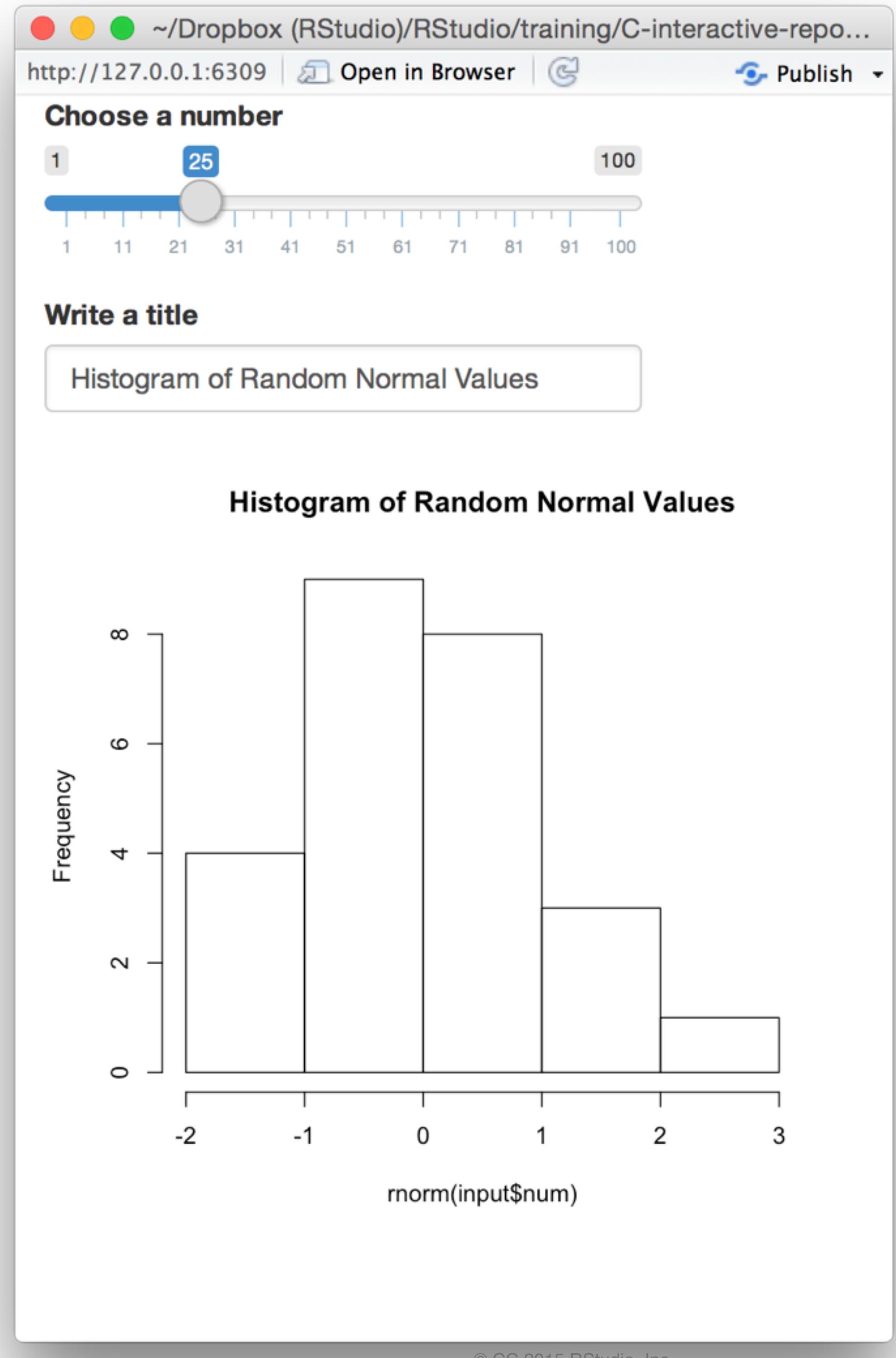
```
# 01-two-inputs

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")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num),
      main = input$title)
  })
}

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



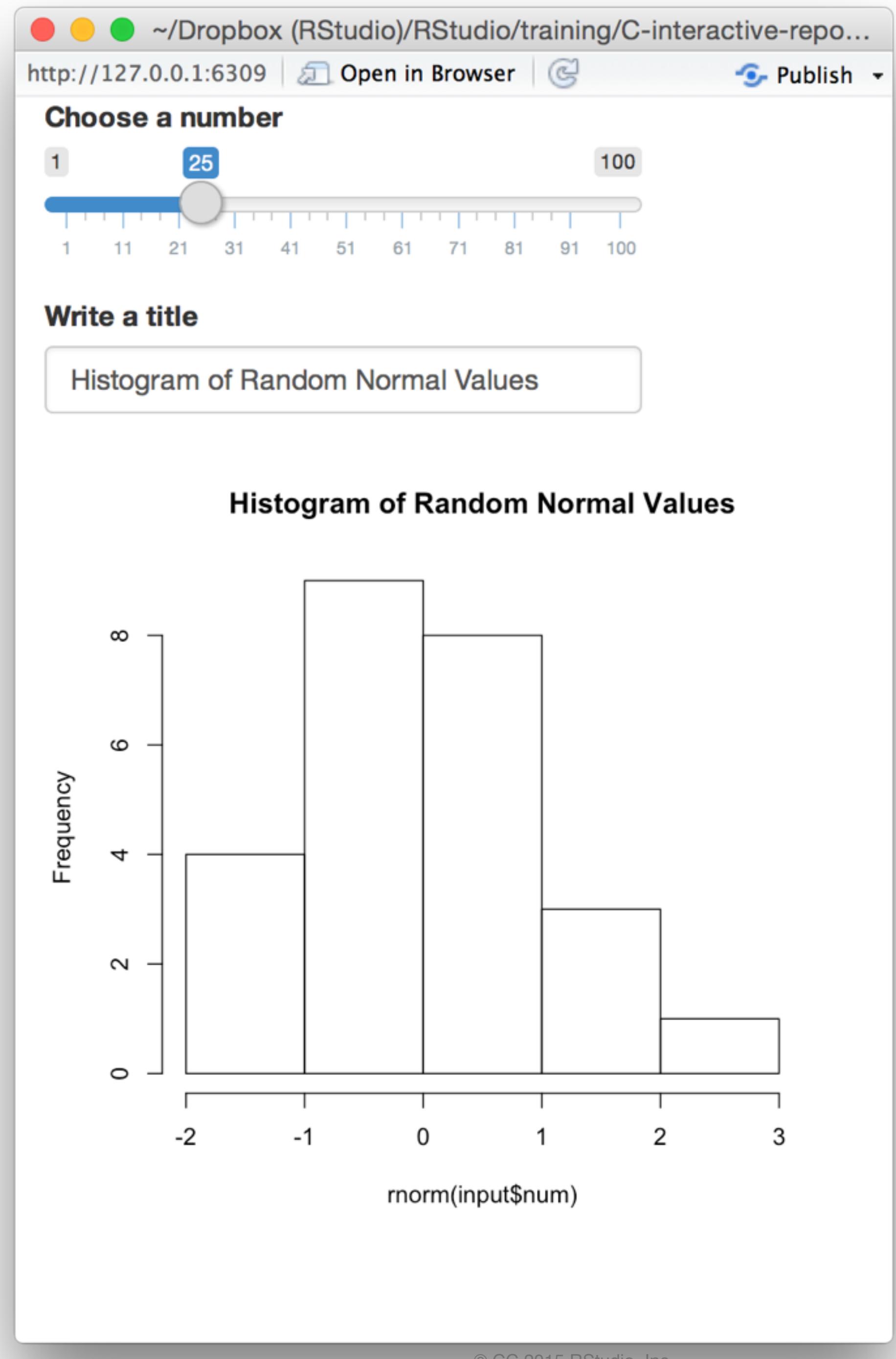
```
# 04-isolate

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")
)

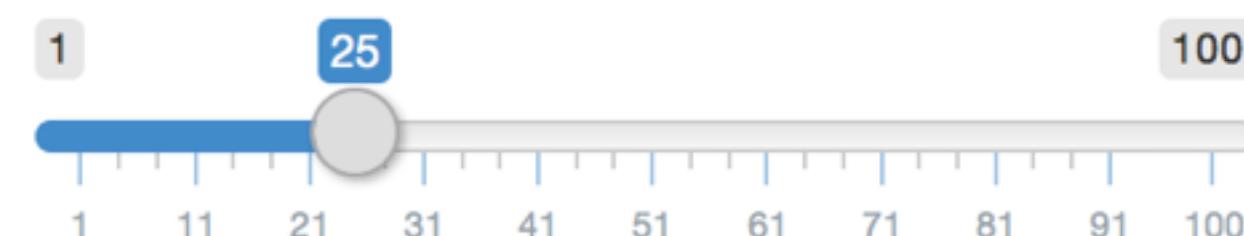
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num),
      main = isolate({input$title}))})
}

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



Choose a number

1 25 100

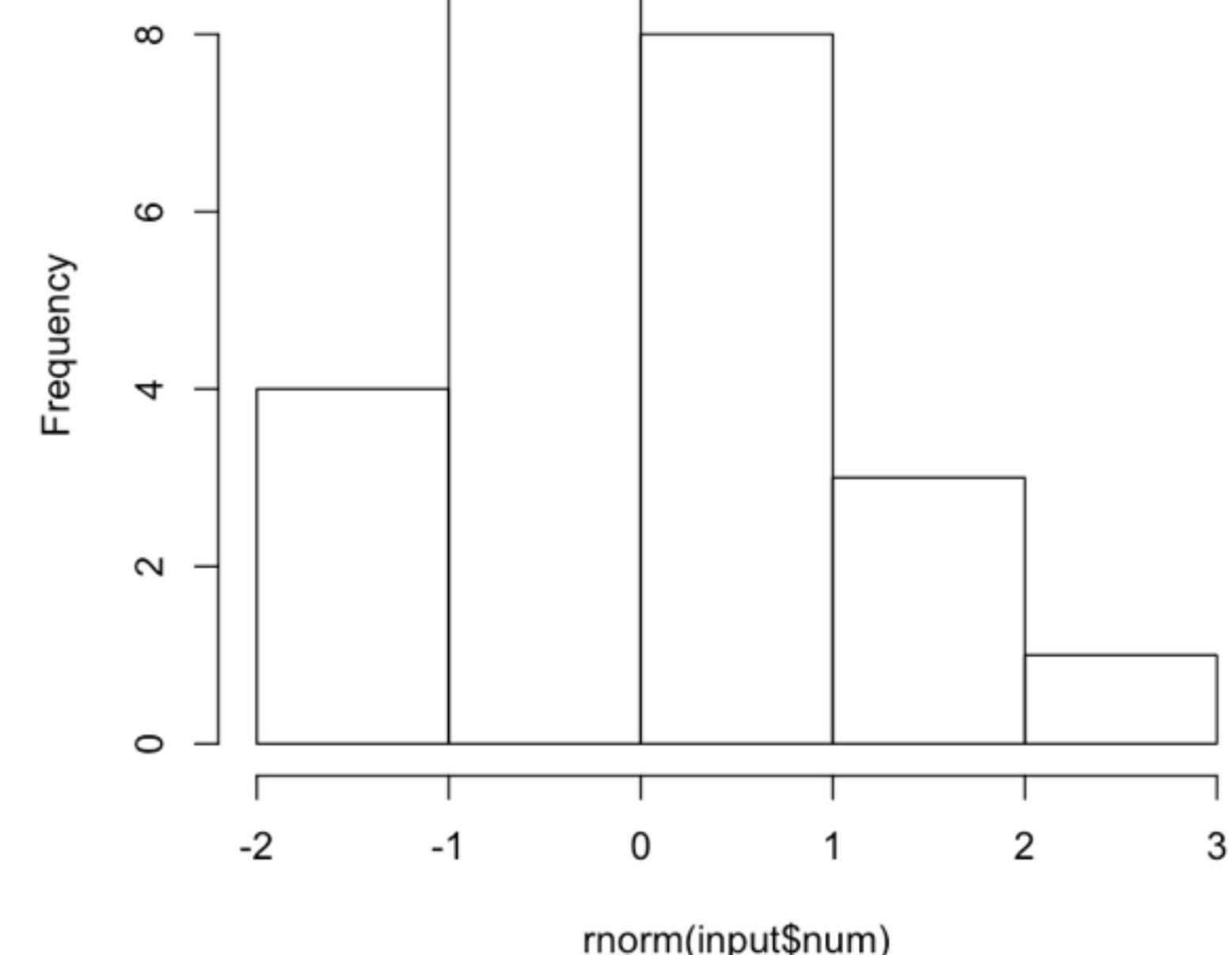


1 11 21 31 41 51 61 71 81 91 100

Write a title

Histogram of Random Normal Values

Histogram of Random Normal Values



input\$num
input\$title

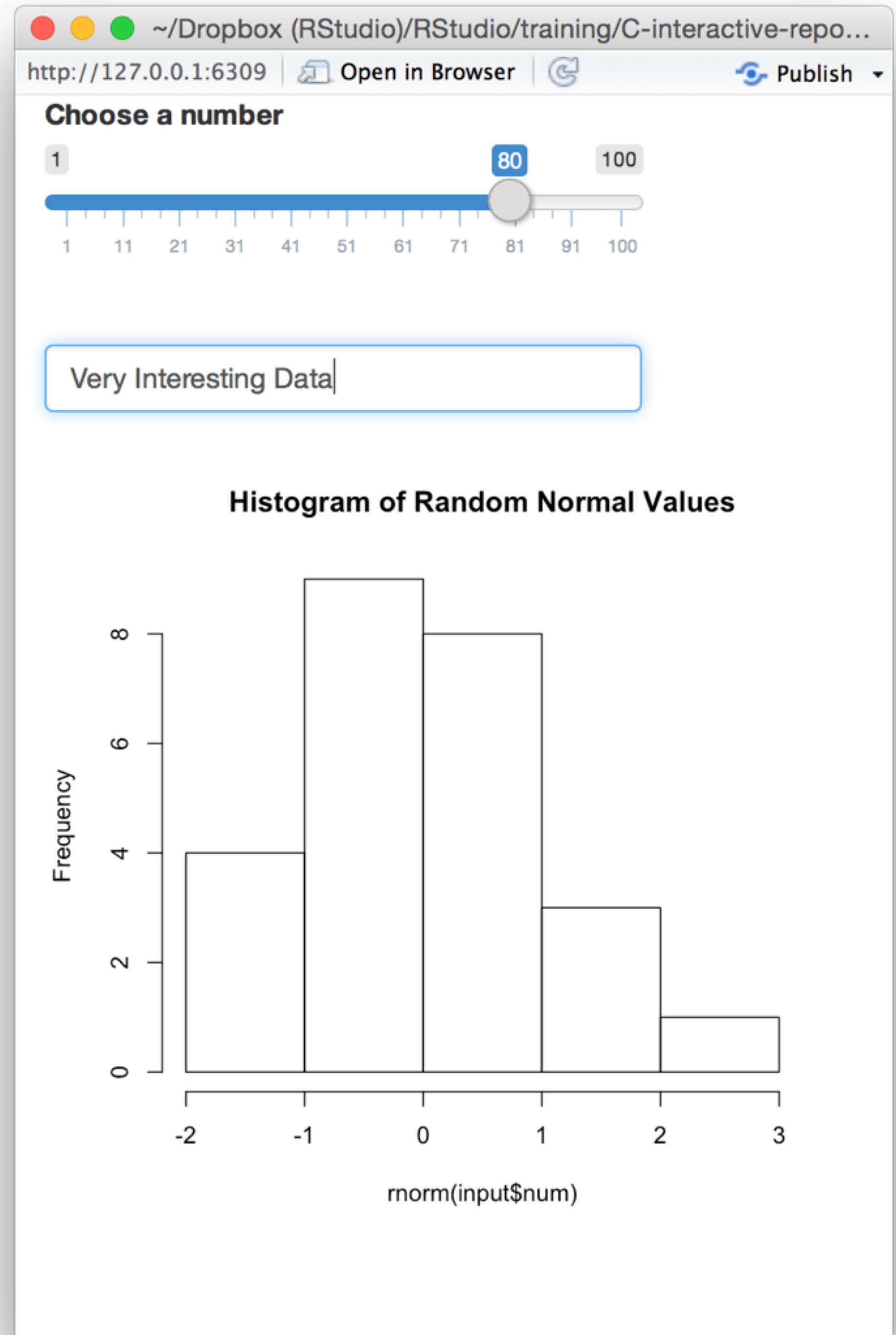
output\$hist <- renderPlot({
 hist(rnorm(input\$num),
 main = isolate(input\$title))
})

input\$num

input\$title

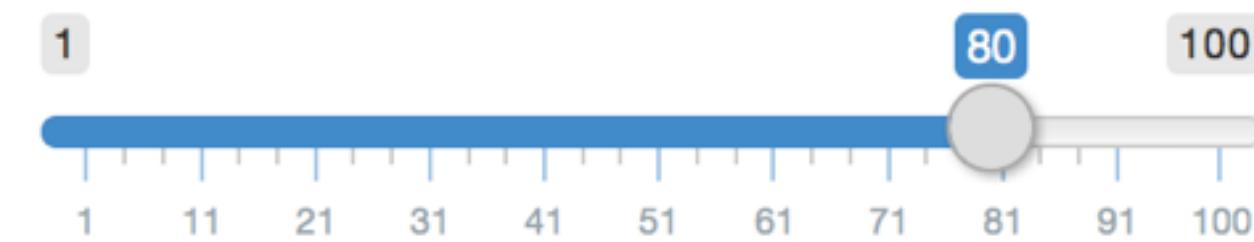


```
output$hist <- renderPlot({  
  hist(rnorm(input$num),  
    main = isolate(input$title))  
})
```



Choose a number

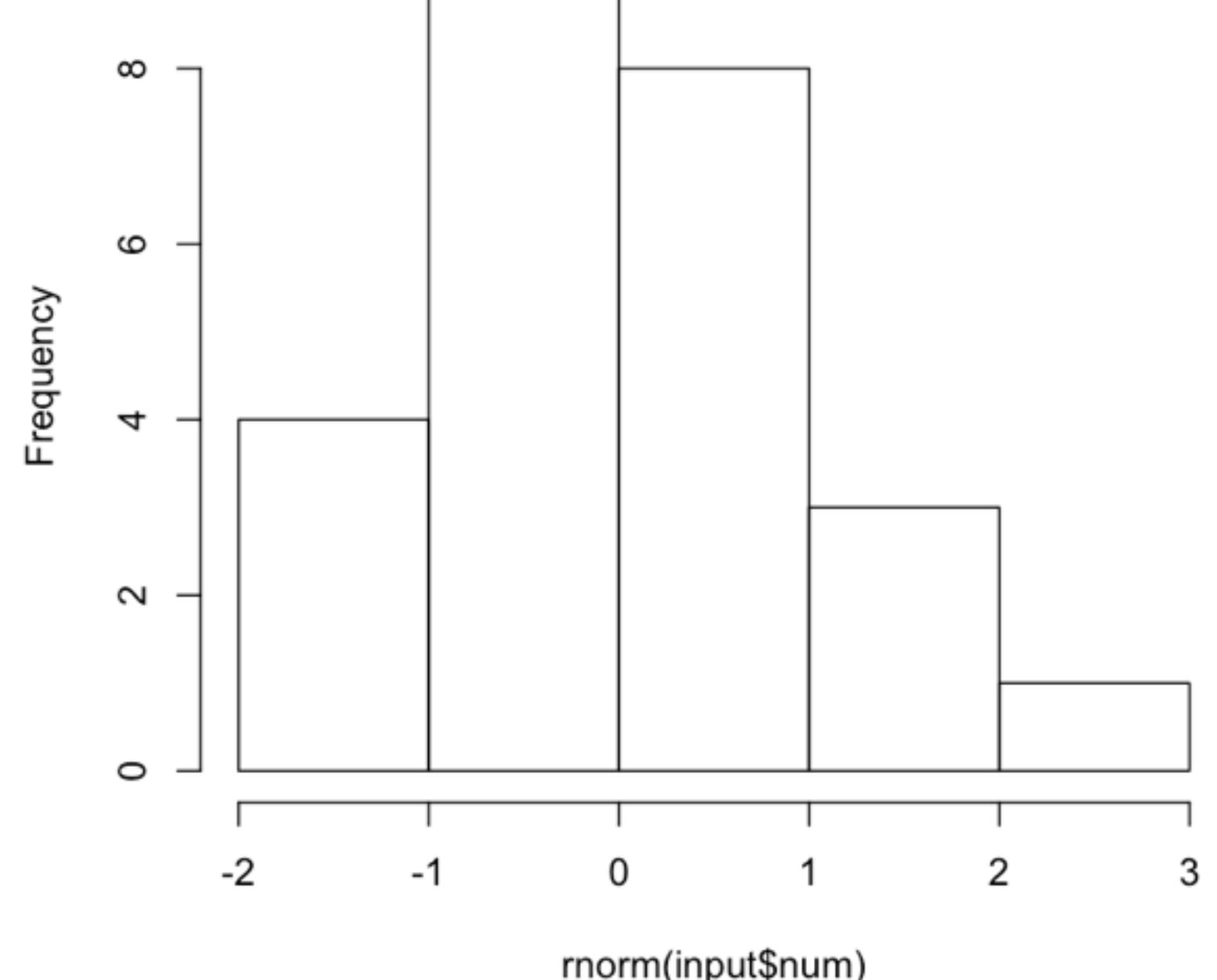
1 80 100



1 11 21 31 41 51 61 71 81 91 100

Very Interesting Data

Histogram of Random Normal Values



```
output$hist <- renderPlot({  
  hist(rnorm(input$num),  
    main = isolate(input$title))  
})
```

Recap: isolate()



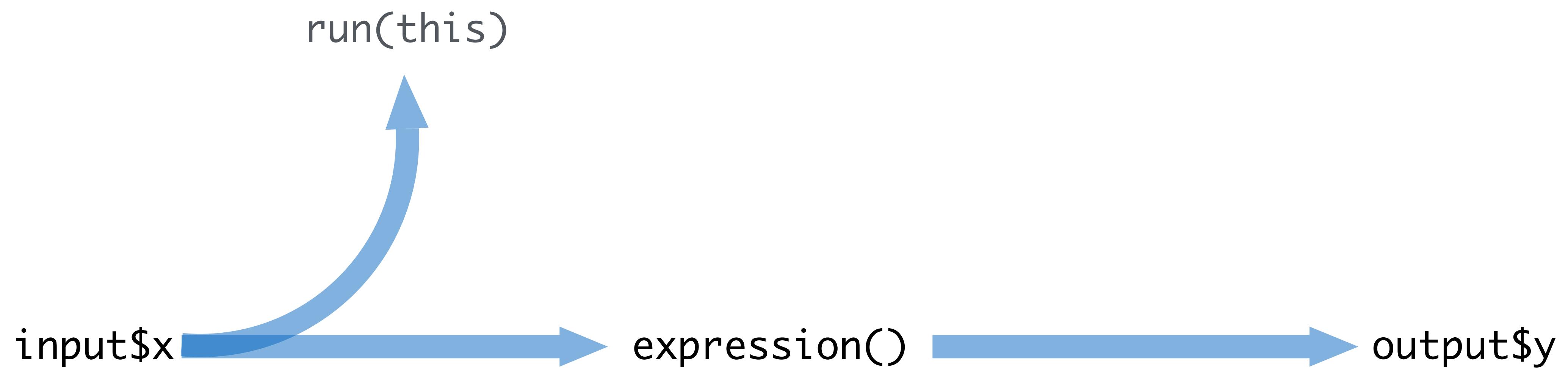
isolate() makes an **non-reactive object**



Use isolate() to treat reactive values like
normal R values

**Trigger code
with observeEvent()**

input\$x  expression  output\$y



Action buttons

An Action Button

Click Me!

input
function

input name
(for internal use)

label to
display

```
actionButton(inputId = "go", label = "Click Me!")
```

Notice:
Id not ID

```
# 05ActionButton
```

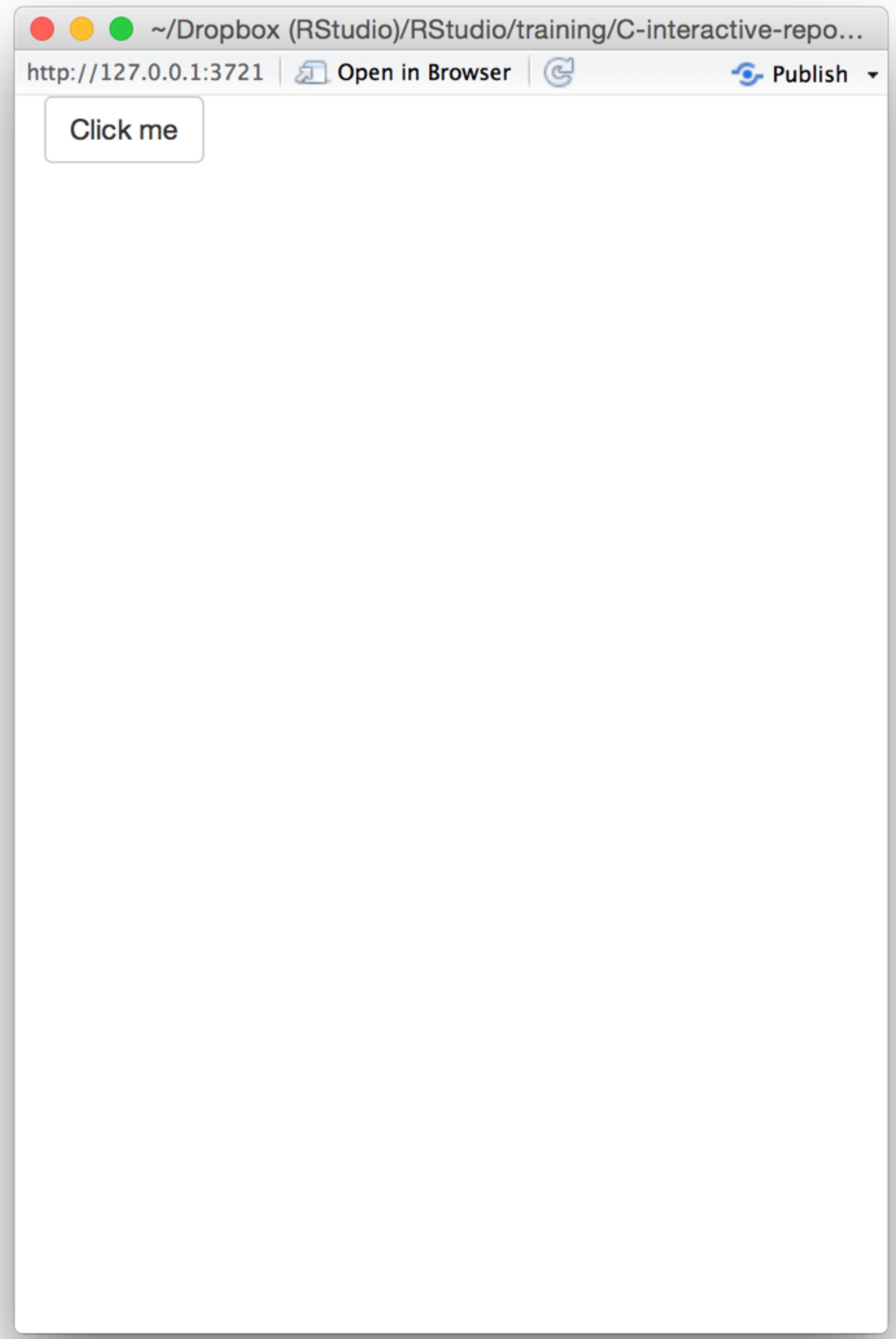
```
library(shiny)
```

```
ui <- fluidPage(  
  actionButton(inputId = "clicks",  
               label = "Click me")  
)
```

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

```
}
```

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



observeEvent()

Triggers code to run on server

```
observeEvent(input$clicks, { print(input$clicks) })
```

reactive value(s) to respond to

(observer invalidates ONLY when this value changes)

code block to run whenever observer is invalidated

note: observer treats this code as if it has been isolated with isolate()

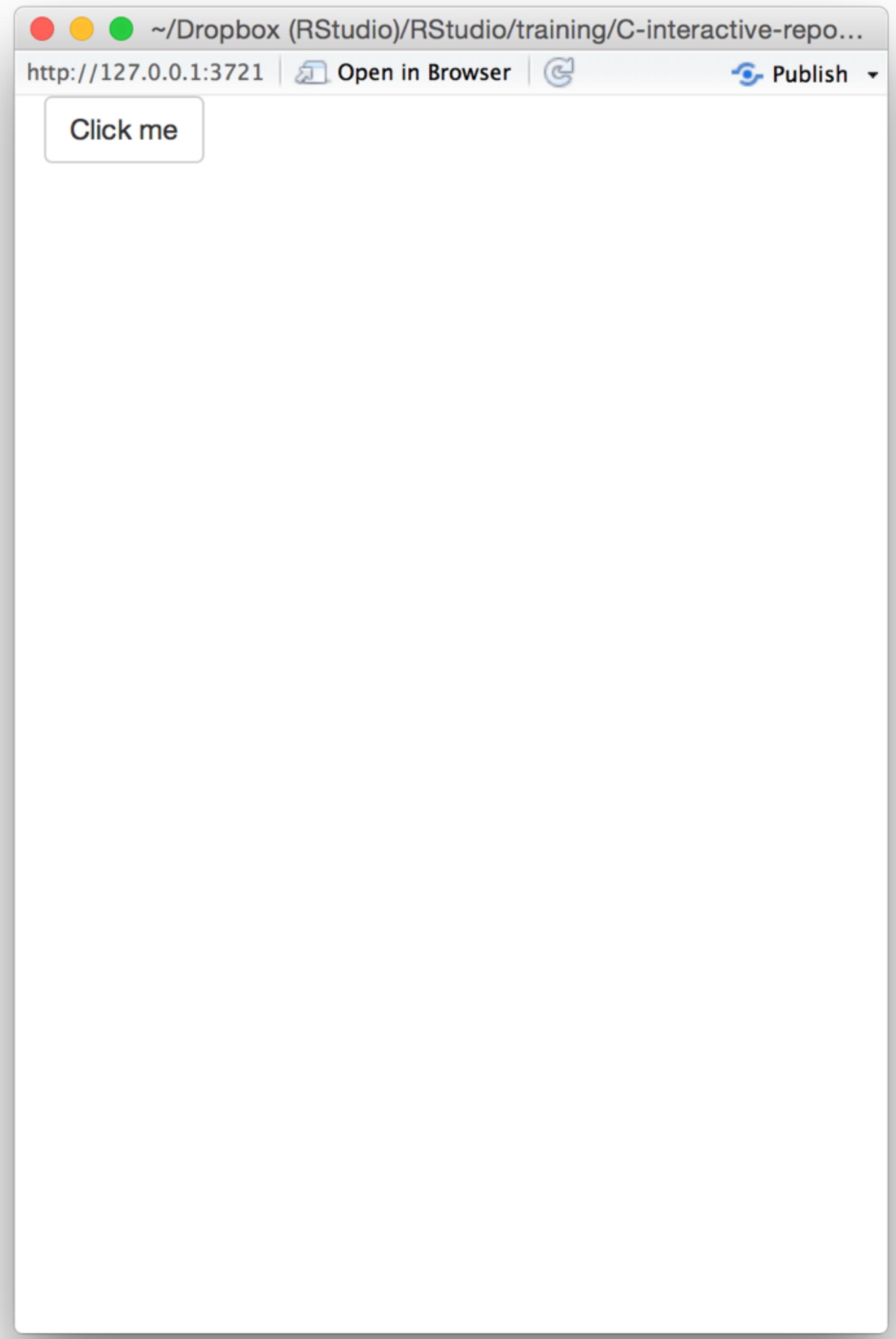
```
# 05ActionButton
```

```
library(shiny)
```

```
ui <- fluidPage(  
  actionButton(inputId = "clicks",  
    label = "Click me")  
)
```

```
server <- function(input, output) {  
  observeEvent(input$clicks, {  
    print(as.numeric(input$clicks))  
  })  
}
```

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



Action buttons article

<http://shiny.rstudio.com/articles/action-buttons.html>

The screenshot shows a web browser window with the title bar "Shiny - Using Action Button X". The address bar contains the URL "shiny.rstudio.com/articles/action-buttons.html". The page itself has a blue header with the "Shiny by RStudio" logo and a search bar. On the left, there's a sidebar with links: OVERVIEW, TUTORIAL, ARTICLES (which is selected and highlighted in blue), GALLERY, REFERENCE, DEPLOY, and HELP. The main content area has a large heading "Using Action Buttons" and a sub-heading "ADDED: 26 MAR 2015". Below this, a paragraph explains that the article describes five patterns for using Shiny's action buttons and action links, noting they are intended for exclusive use with `observeEvent()` or `eventReactive()`. A section titled "How action buttons work" follows, with a note about creating them with `actionButton()` and `actionLink()`, each taking two arguments: `inputId` and `label`. Below this is a code block:

```
actionButton("button", "An action button")
actionLink("button", "An action link")
```

Further down, a callout box contains the text "An action button appears as a button in your app." with a sample button labeled "An action button". At the bottom, another callout box states "An action link appears as a hyperlink, but behaves in the same way as an action button."

observe()

Also triggers code to run on server.

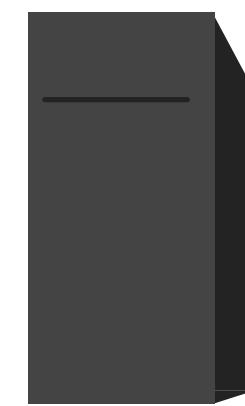
Uses same syntax as render*, reactive(), and isolate()

```
observe({ print(input$clicks) })
```

observer will respond to
*every reactive value in the
code*

code block to run
whenever observer is
invalidated

Recap: observeEvent()



observeEvent() **triggers code to run on the server**

```
observeEvent(input$clicks, { print(input$clicks) })
```

reactive value(s)
to respond to

A snippet of R code showing an observeEvent call. A callout box points from the text "reactive value(s) to respond to" to the "input\$clicks" argument in the code.

Specify precisely which reactive values should invalidate the observer

observe()

Use observe() for a more implicit syntax

Delay reactions with eventReactive()

```
# 07-eventReactive

library(shiny)

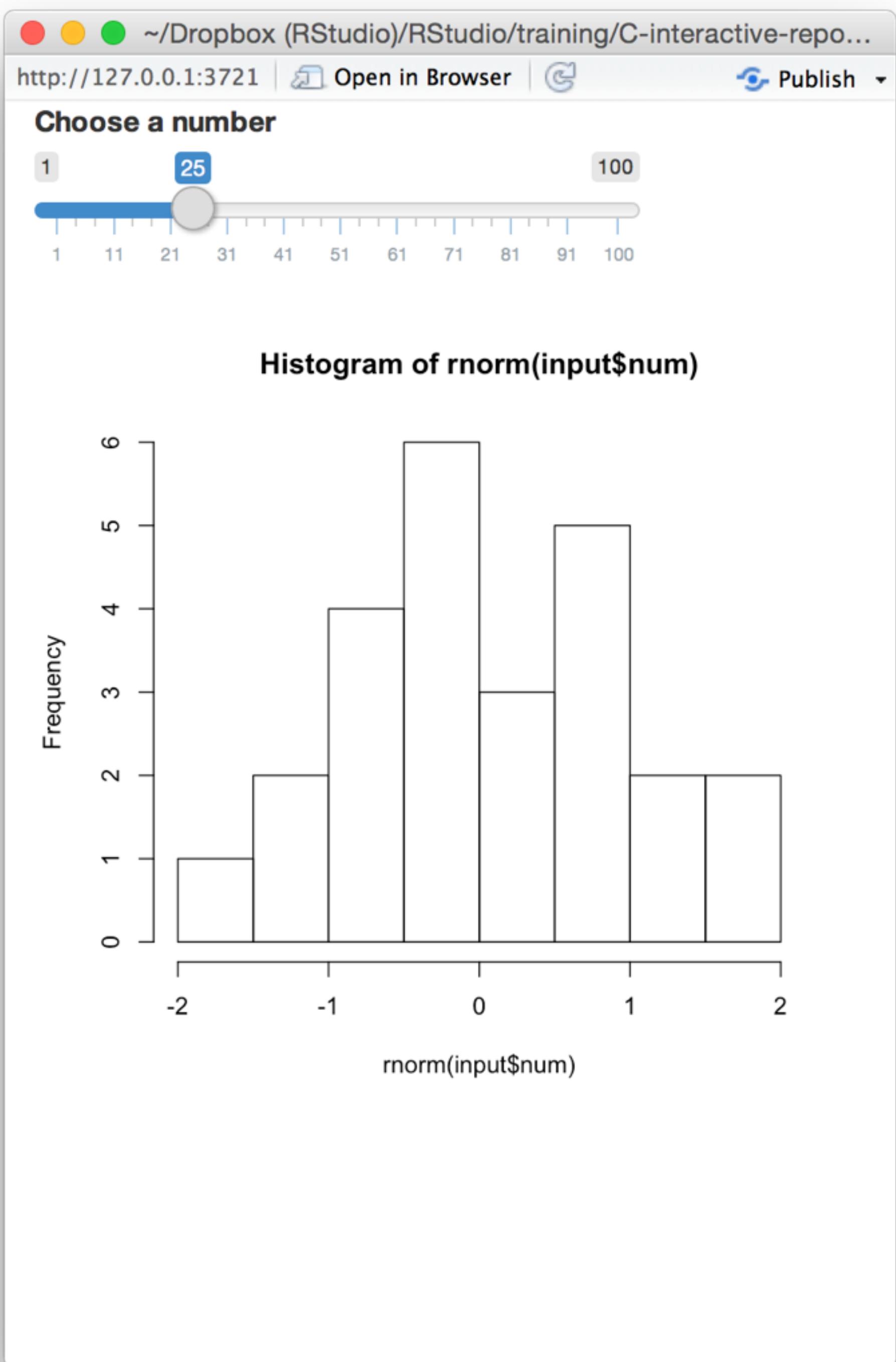
ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),

  plotOutput("hist")
)

server <- function(input, output) {

  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
}

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



```
# 07-eventReactive

library(shiny)

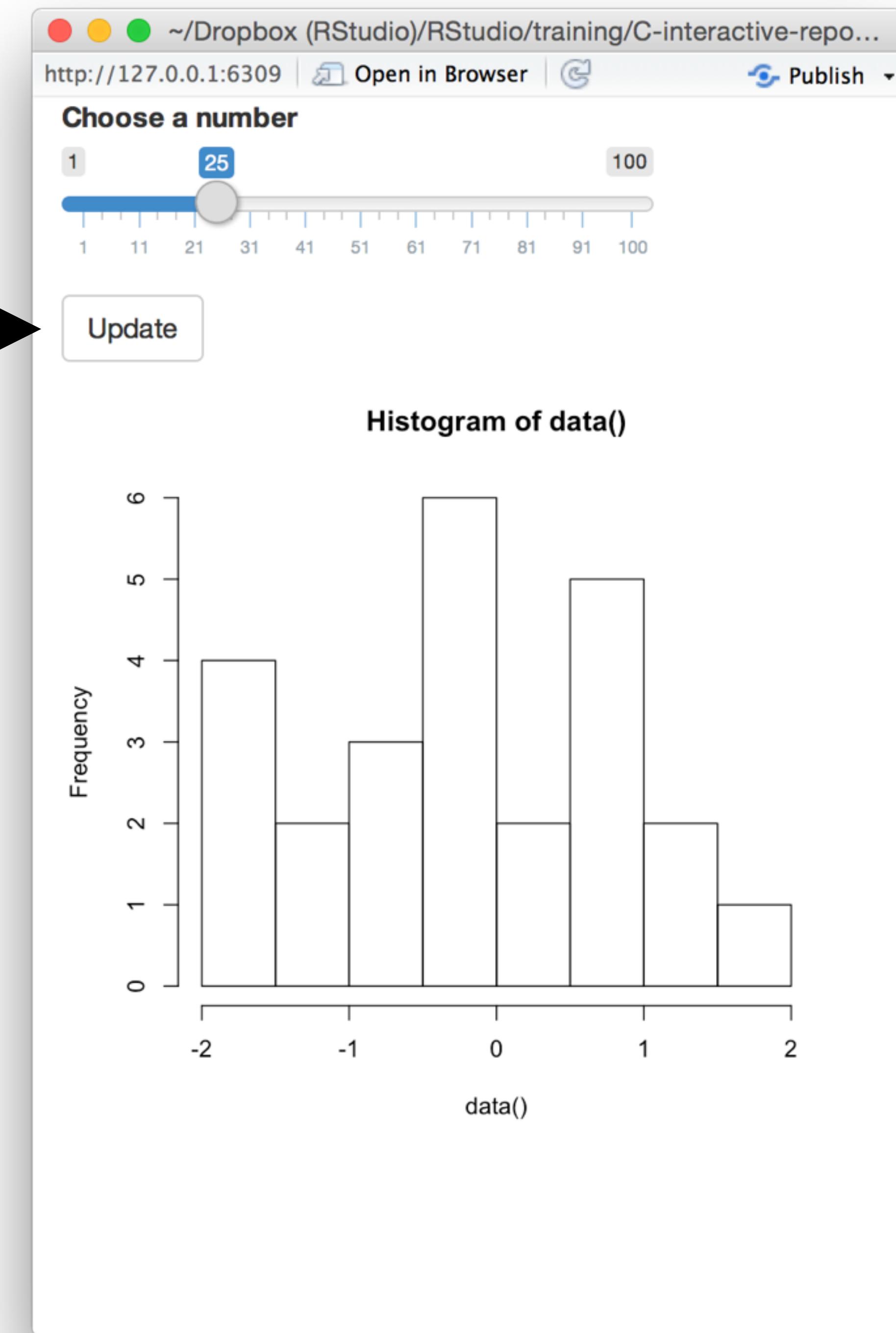
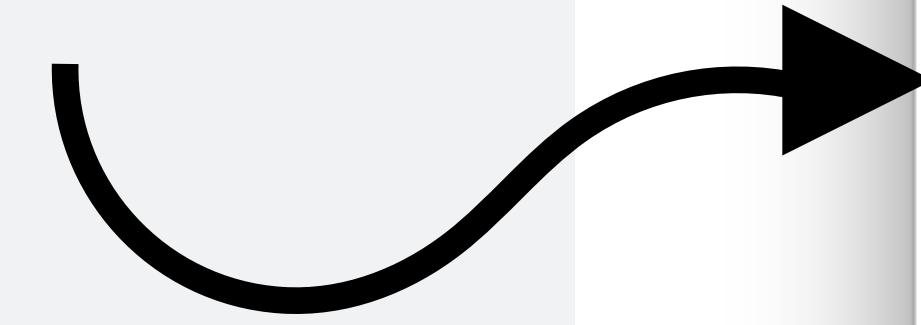
ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  actionButton(inputId = "go",
    label = "Update"),
  plotOutput("hist")
)

server <- function(input, output) {

  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
}

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

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



eventReactive()

A reactive expression that only responds to specific values

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

reactive value(s) to respond to

code used to build (and rebuild) object

note: expression treats this code as if it has been isolated with isolate()

(expression invalidates ONLY when this value changes)

```
# 07-eventReactive

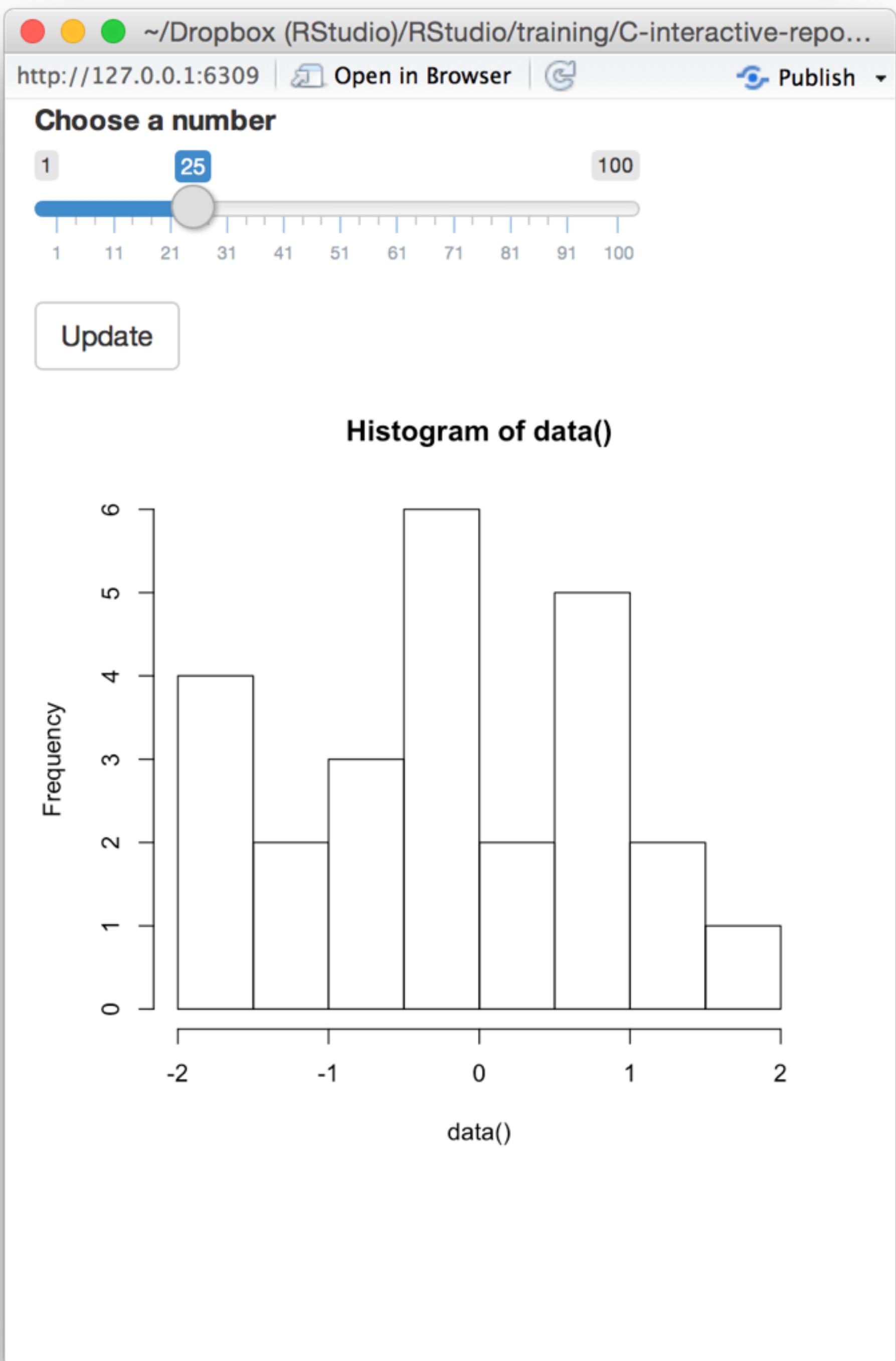
library(shiny)

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) {

  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
}

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



```
# 07-eventReactive

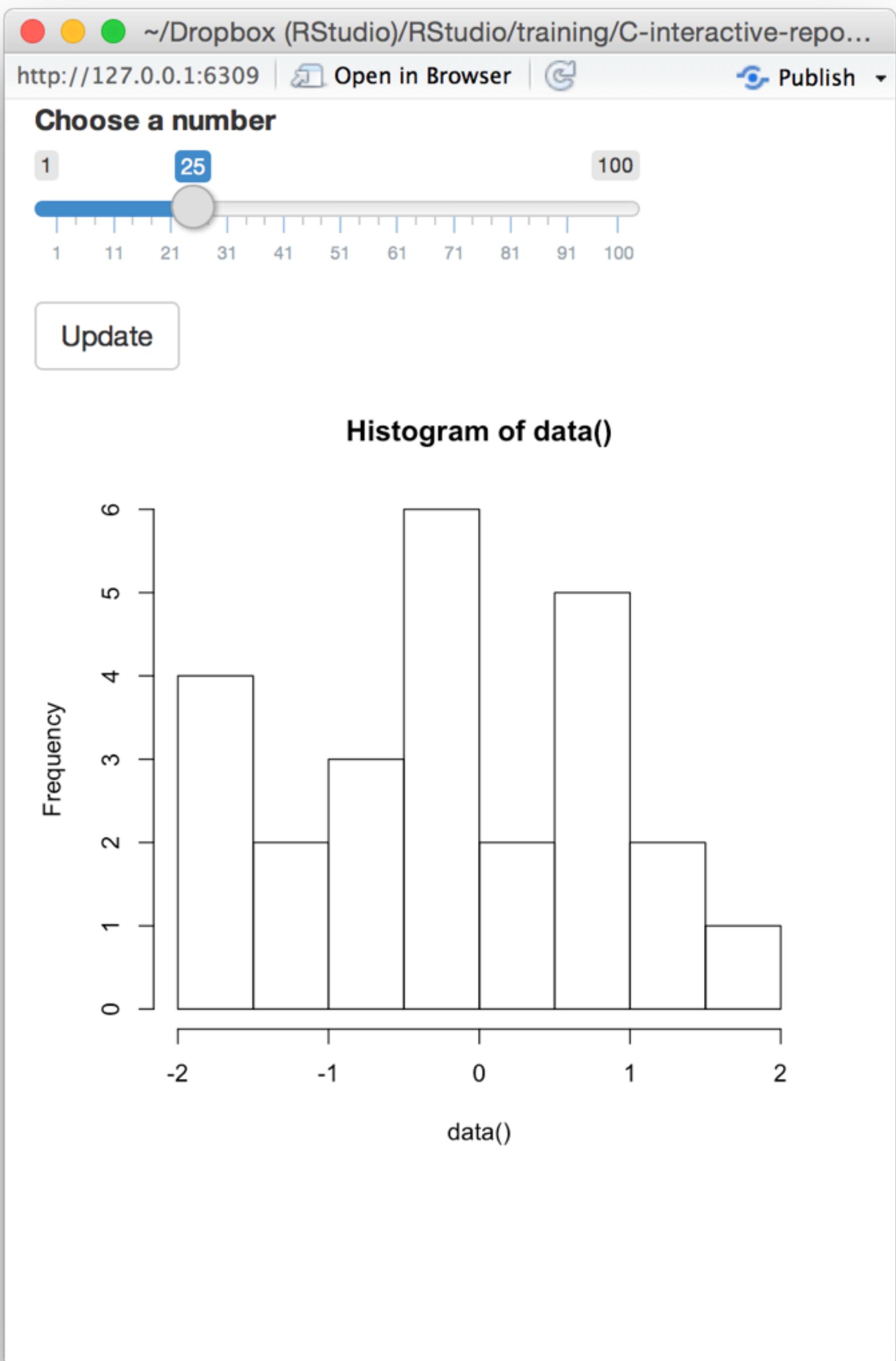
library(shiny)

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

  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
}

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



```
# 07-eventReactive

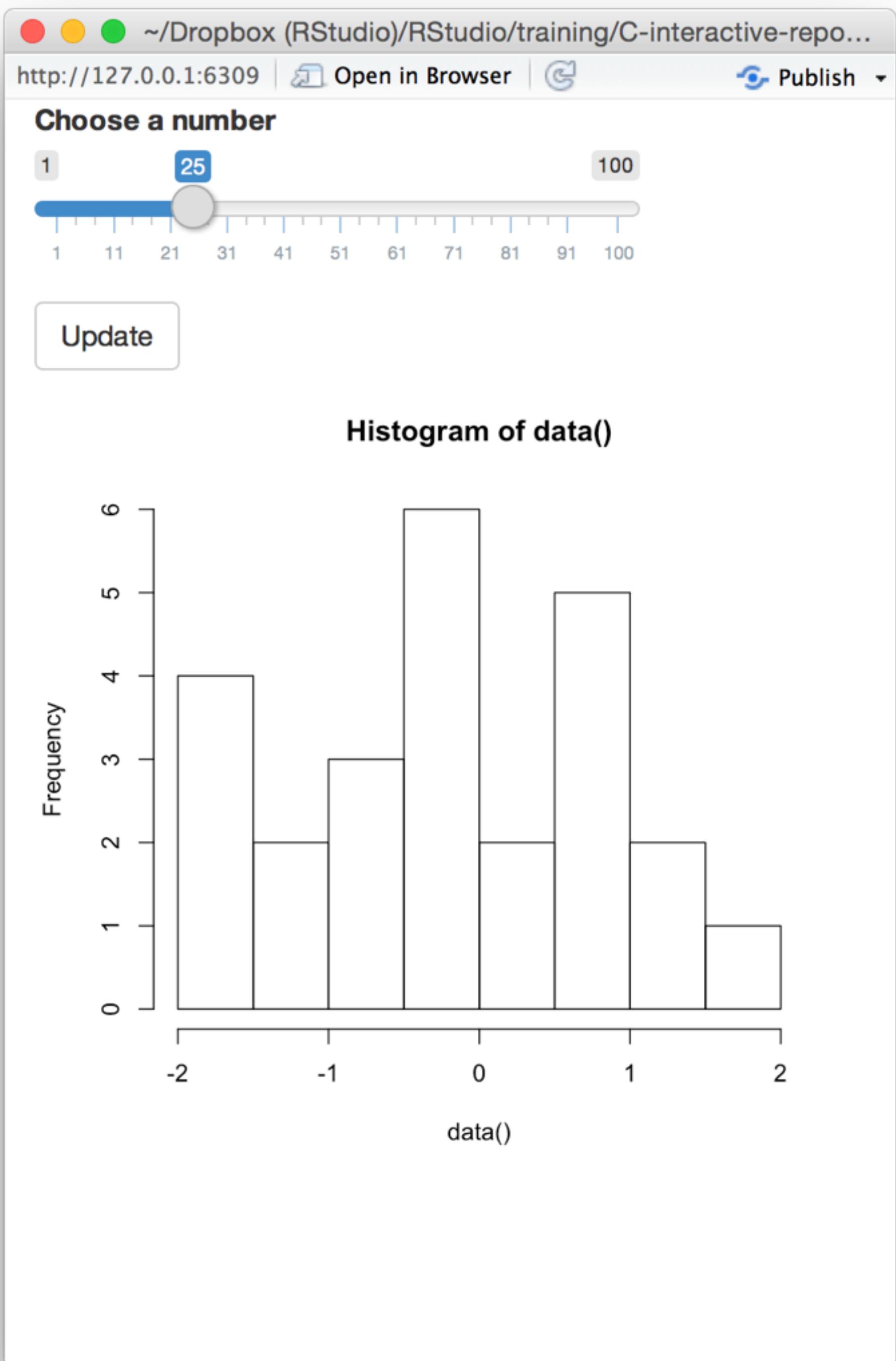
library(shiny)

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

  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
}

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



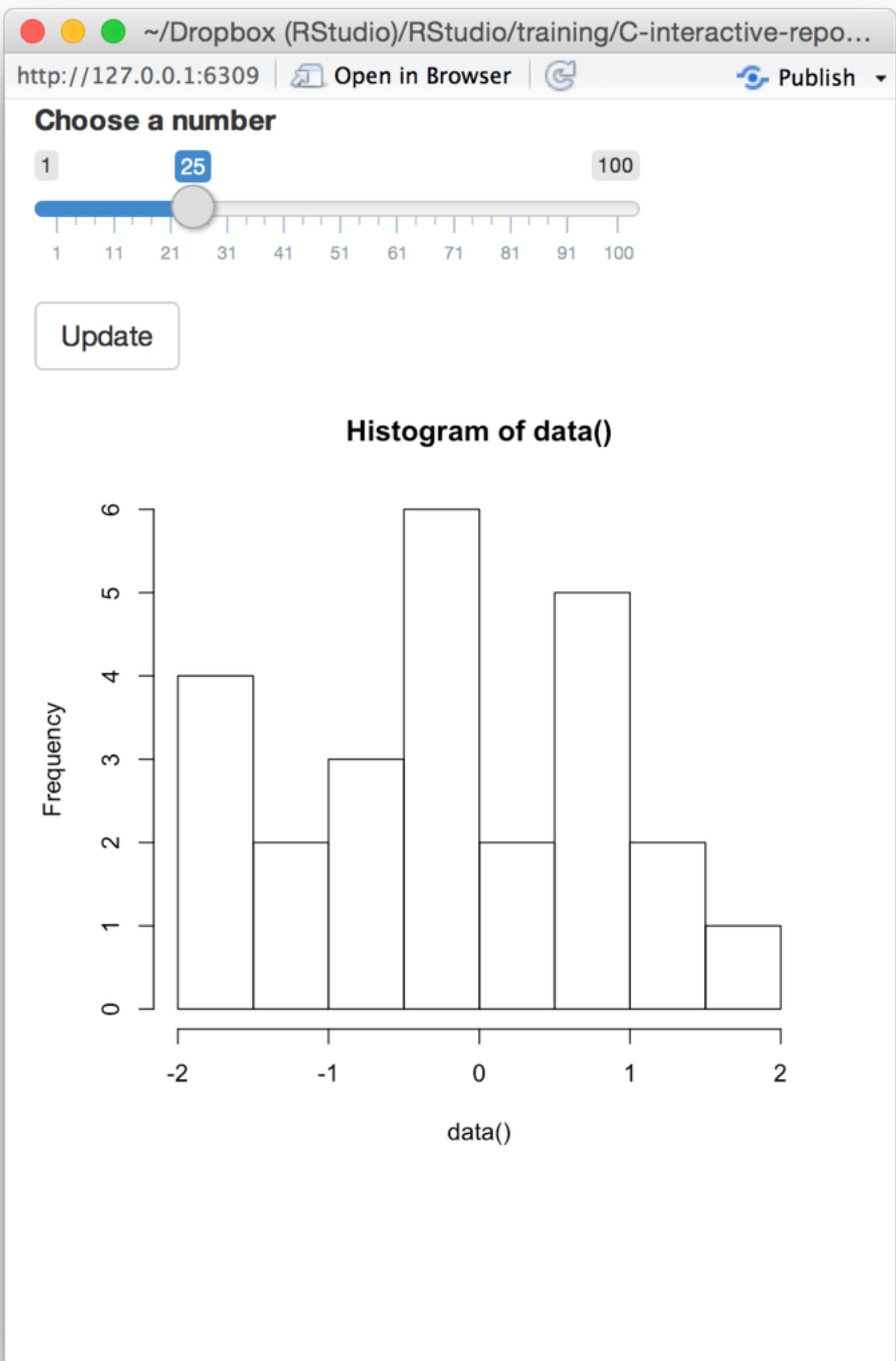
```
# 07-eventReactive

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

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



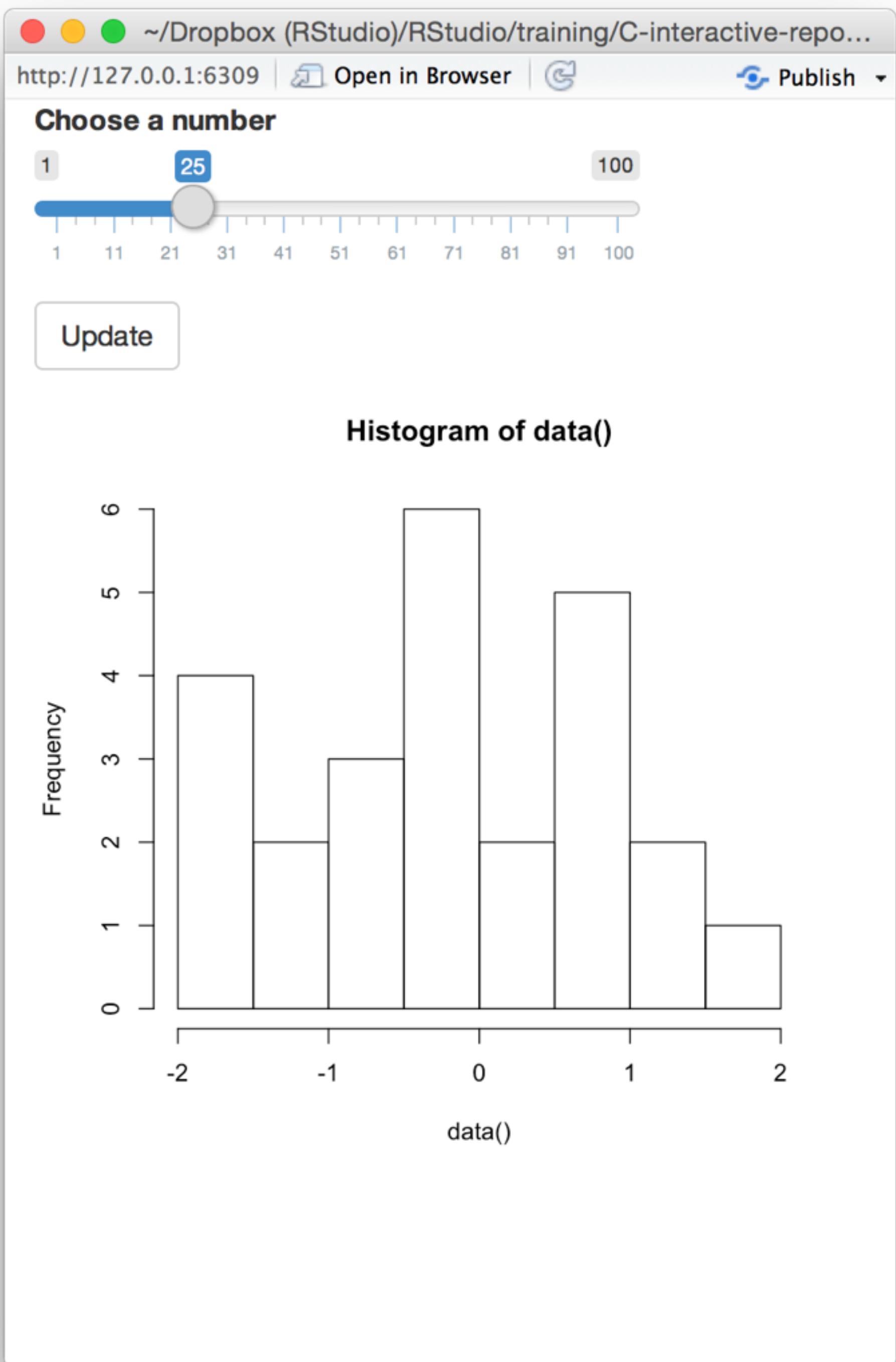
```
# 07-eventReactive

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

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

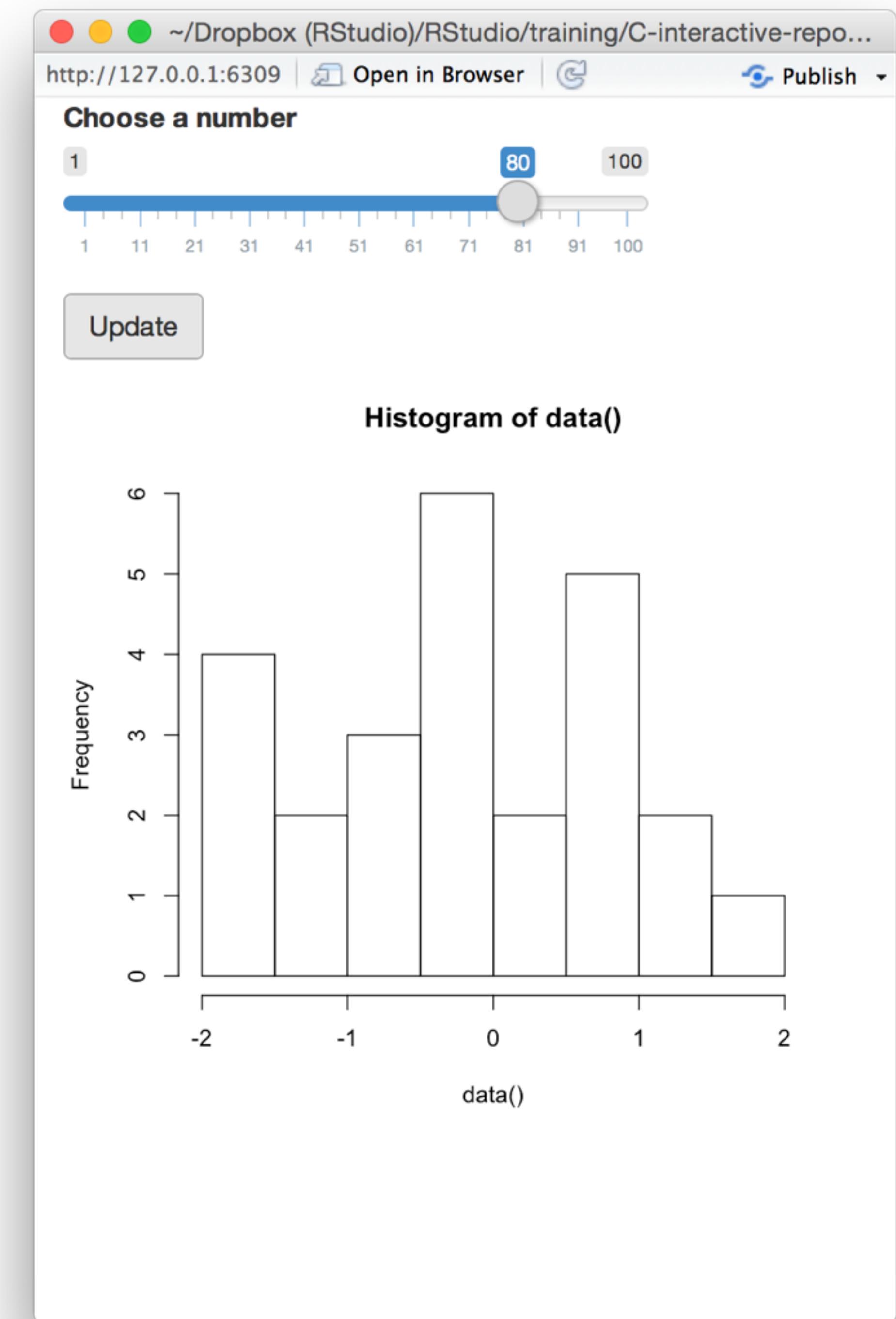


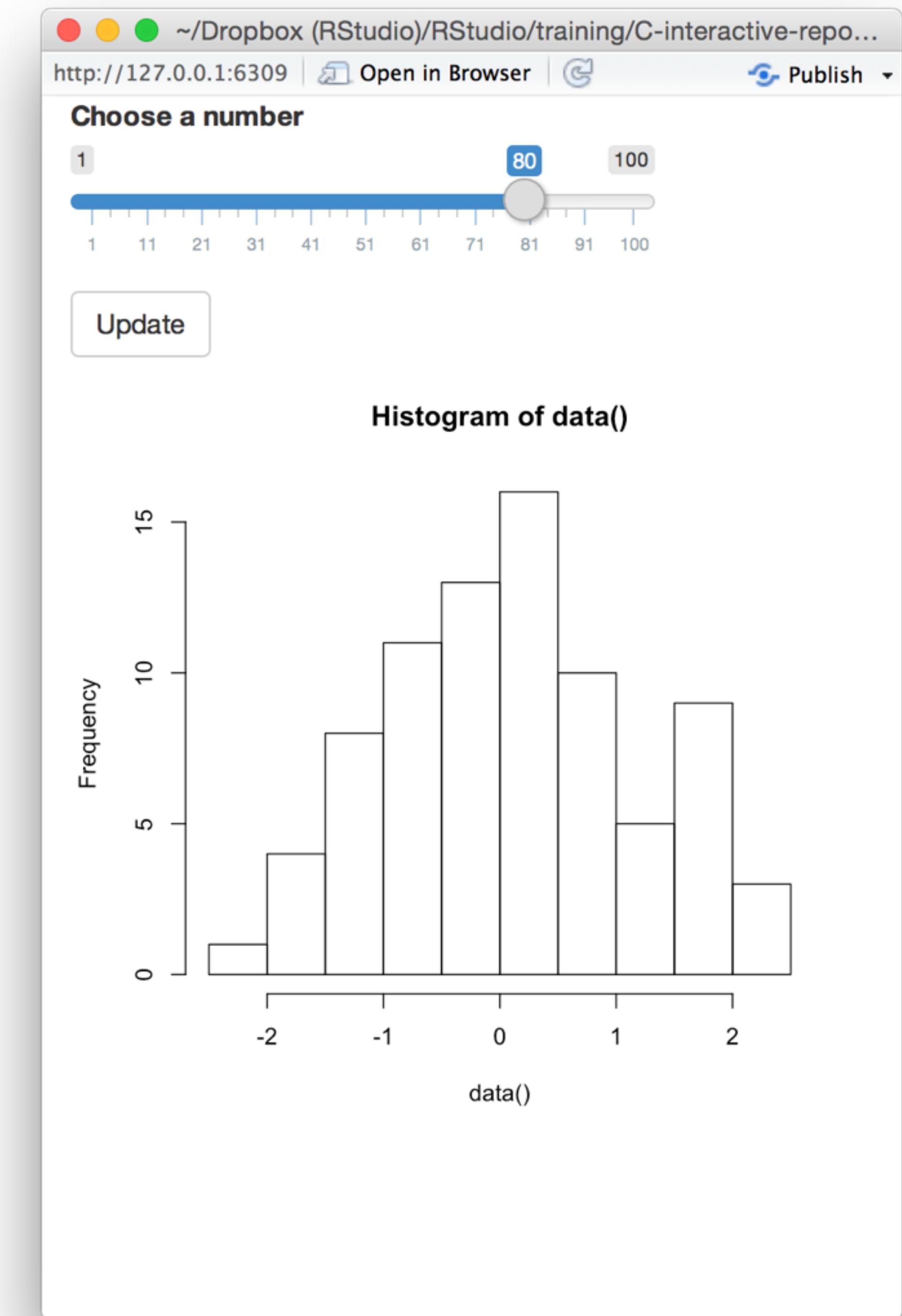
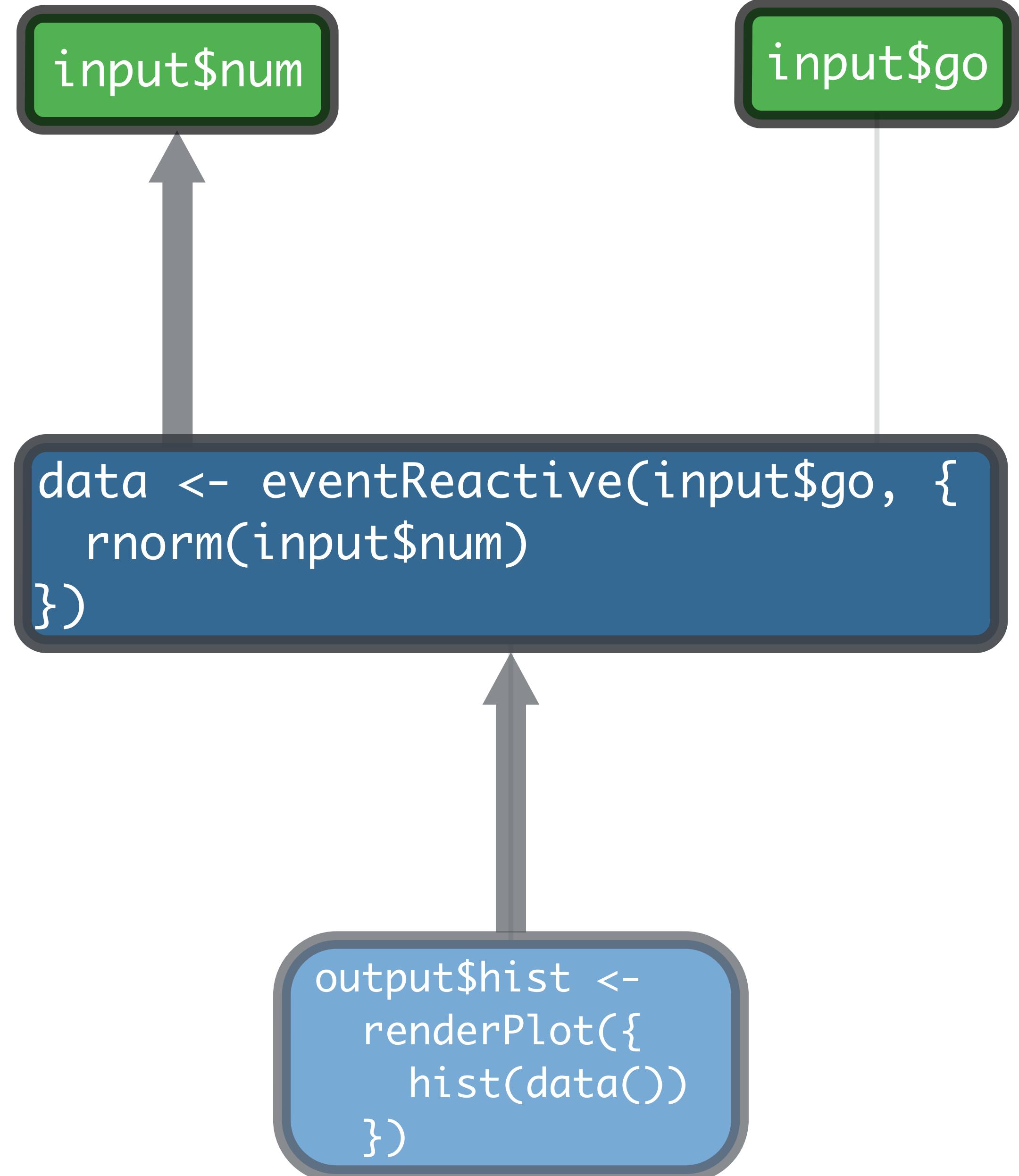
```
input$num
```

```
input$go
```

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

```
output$hist <-  
  renderPlot({  
    hist(data())  
})
```





Recap: eventReactive()

Update

Use `eventReactive()` to **delay reactions**

`data()`

`eventReactive()` creates a **reactive expression**

```
eventReactive(input$go, { rnorm(input$num) })
```

reactive value(s)
to respond to

You can specify **precisely** which reactive values should invalidate the expression

Manage state with reactiveValues()

Input values

The input value changes whenever a user changes the input.

Choose a number

A slider input with a title "Choose a number". The slider has a blue track and a grey handle. The value "25" is displayed in a blue box above the slider. The slider scale ranges from 1 to 100 with major ticks every 10 units and minor ticks every 2 units. The handle is positioned between 21 and 31.

input\$num = 25

Choose a number

A slider input with a title "Choose a number". The slider has a blue track and a grey handle. The value "50" is displayed in a blue box above the slider. The slider scale ranges from 1 to 100 with major ticks every 10 units and minor ticks every 2 units. The handle is positioned between 41 and 51.

input\$num = 50

Choose a number

A slider input with a title "Choose a number". The slider has a blue track and a grey handle. The value "75" is displayed in a blue box above the slider. The slider scale ranges from 1 to 100 with major ticks every 10 units and minor ticks every 2 units. The handle is positioned between 71 and 81.

input\$num = 75

You cannot set these
values in your code

reactiveValues()

Creates a list of reactive values to manipulate programmatically

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

(optional) elements
to add to the list

```

# 08-reactiveValues

library(shiny)

ui <- fluidPage(
  actionButton(inputId = "norm", label = "Normal"),
  actionButton(inputId = "unif", label = "Uniform"),
  plotOutput("hist")
)

server <- function(input, output) {

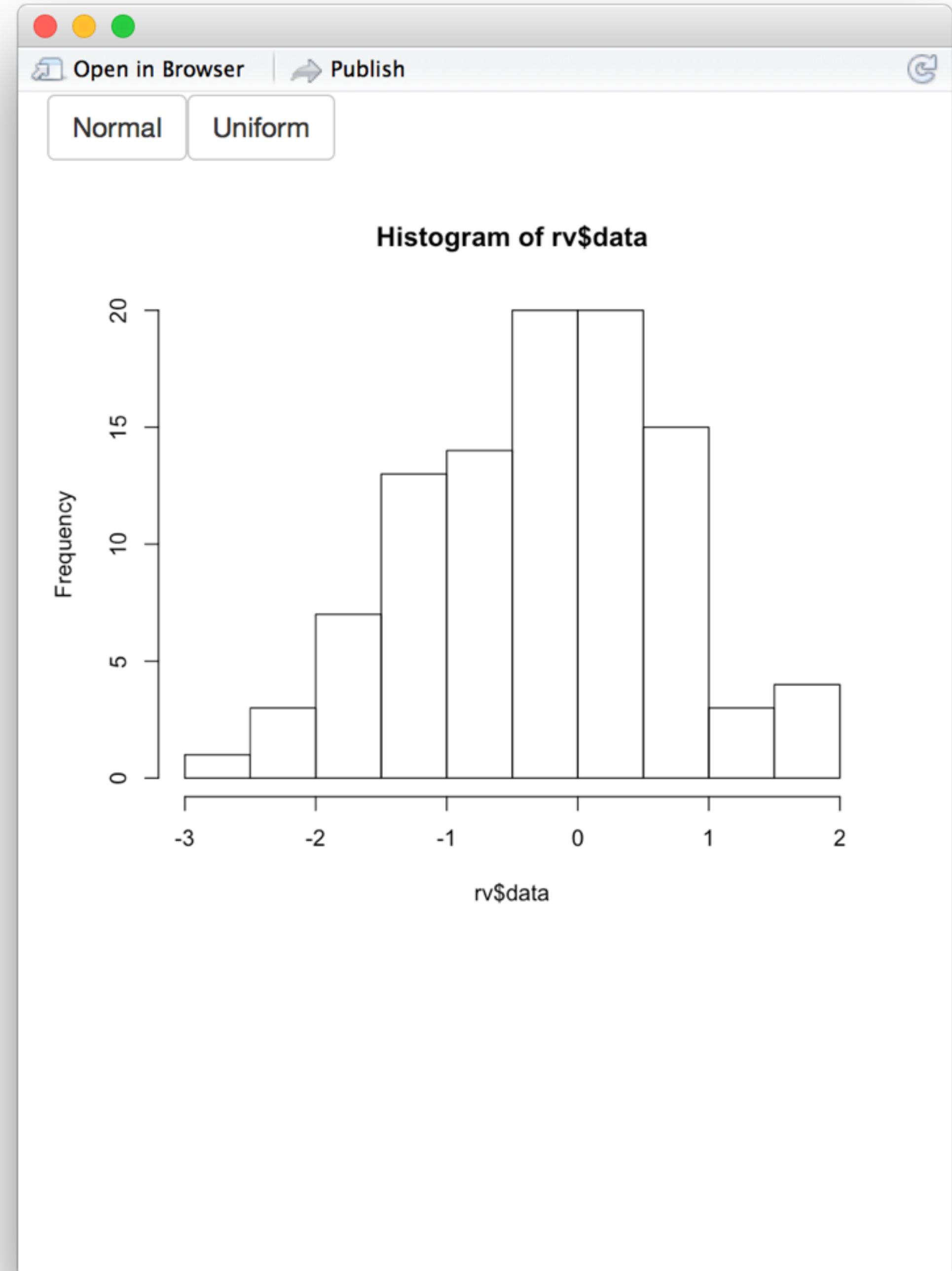
  rv <- reactiveValues(data = rnorm(100))

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

  output$hist <- renderPlot({
    hist(rv$data)
  })
}

shinyApp(ui = ui, server = server)

```

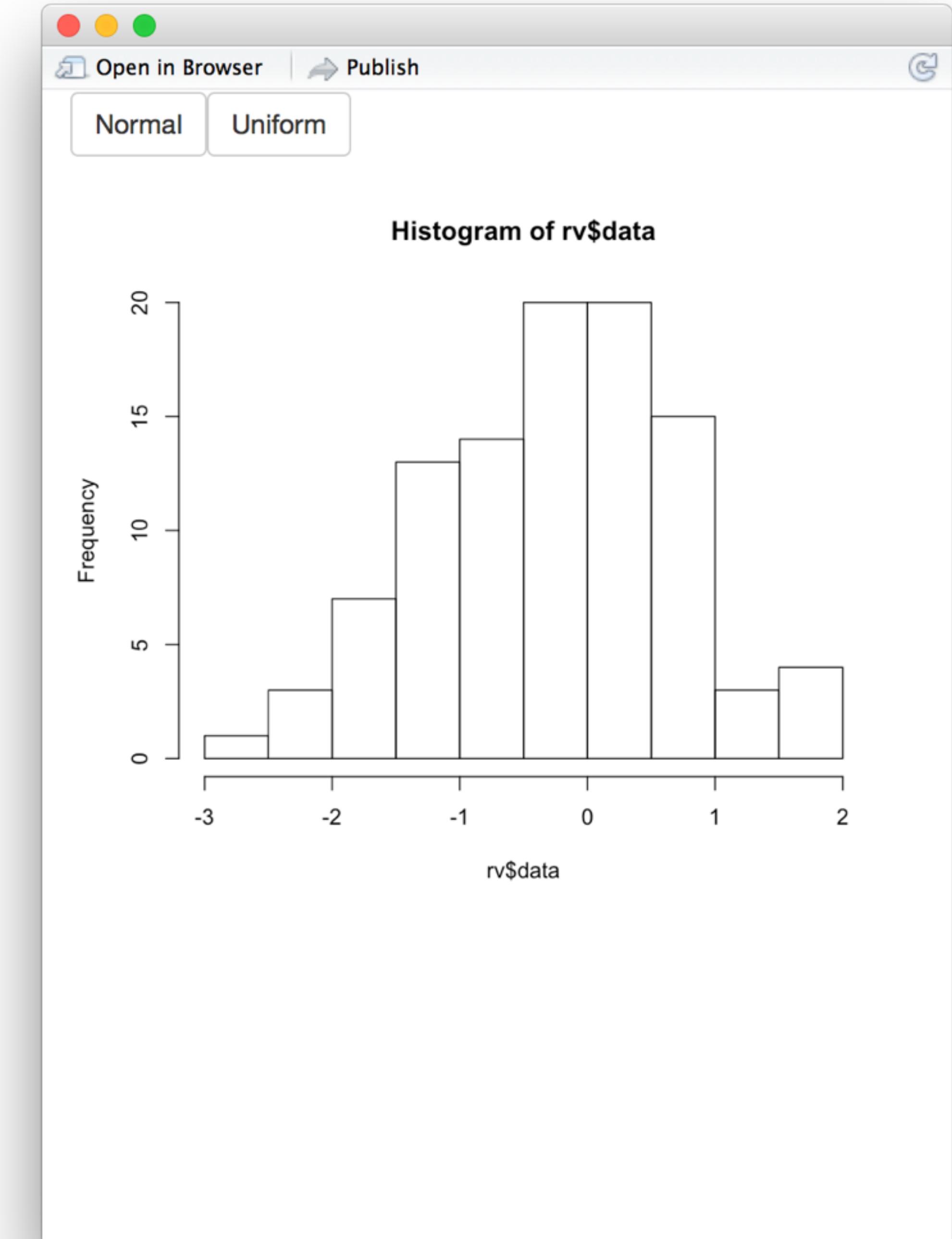


```
input$norm
```

```
rv$data  
rnorm(100)
```

```
input$unif
```

```
output$hist <-  
renderPlot({  
  hist(rv$data)  
})
```

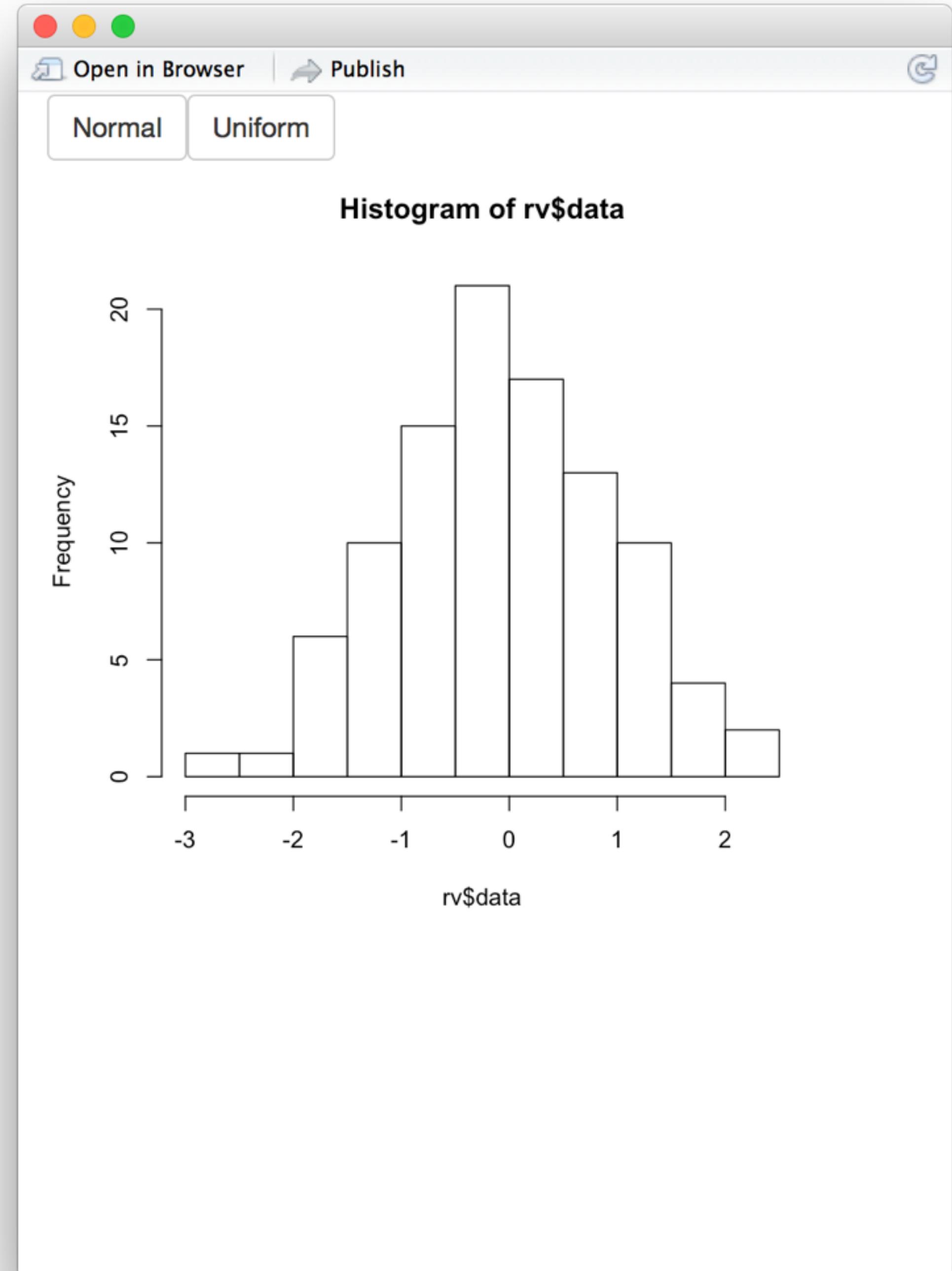


input\$norm

rv\$data
rnorm(100)

input\$unif

```
output$hist <-  
  renderPlot({  
    hist(rv$data)  
  })
```

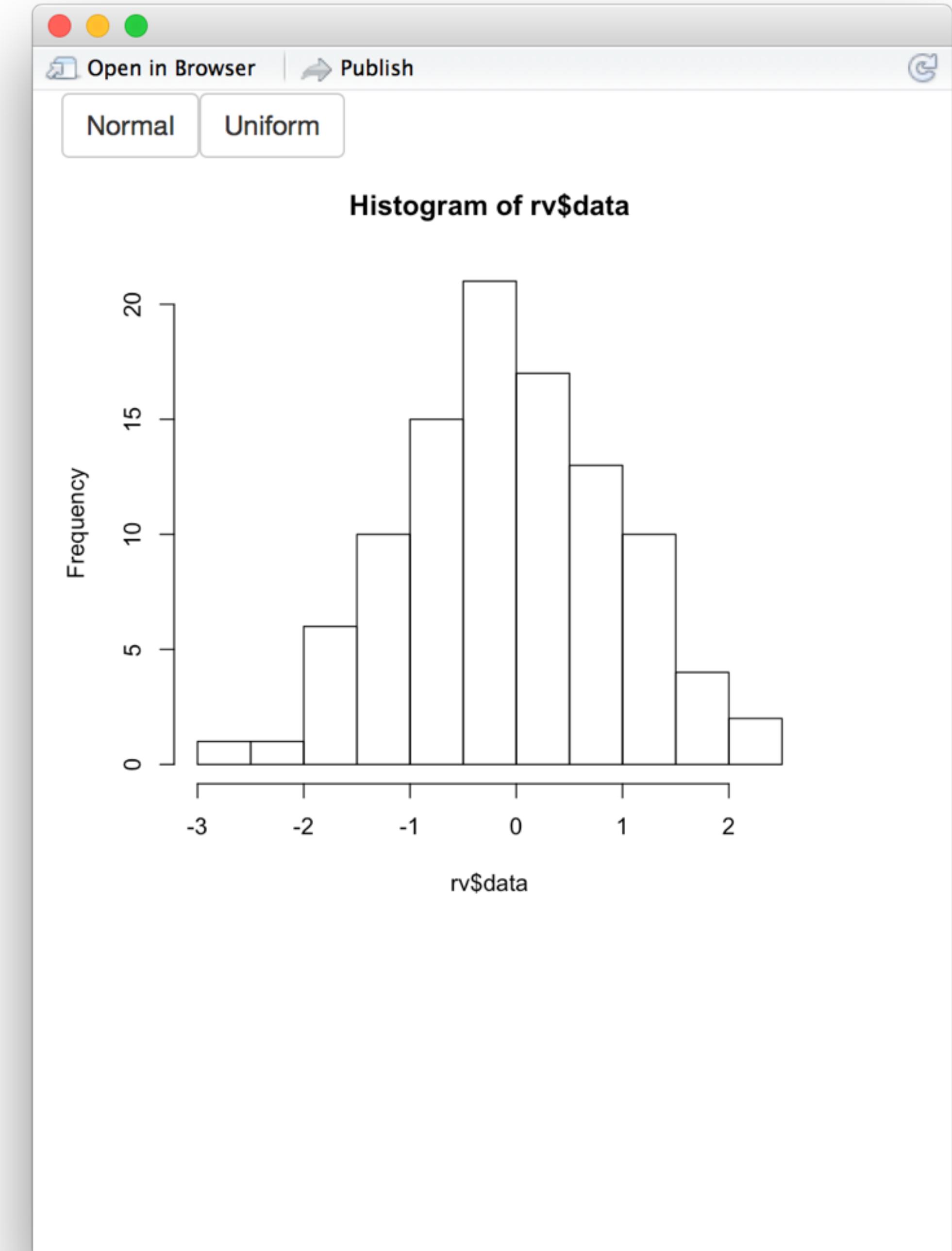


```
input$norm
```

```
rv$data  
runif(100)
```

```
input$unif
```

```
output$hist <-  
renderPlot({  
  hist(rv$data)  
})
```

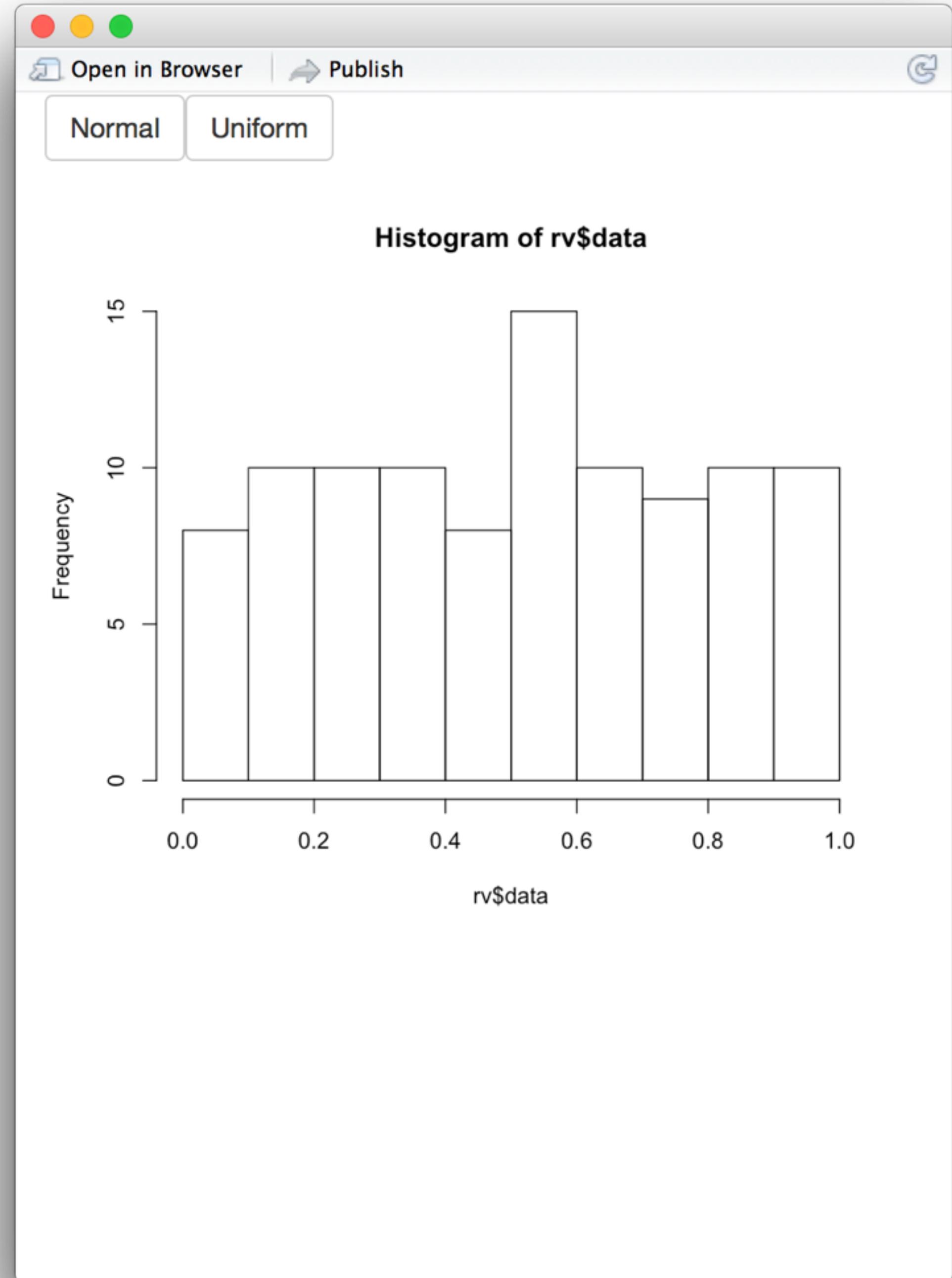


input\$norm

rv\$data
runif(100)

input\$unif

```
output$hist <-  
  renderPlot({  
    hist(rv$data)  
  })
```



```

# 08-reactiveValues

library(shiny)

ui <- fluidPage(
  actionButton(inputId = "norm", label = "Normal"),
  actionButton(inputId = "unif", label = "Uniform"),
  plotOutput("hist")
)

server <- function(input, output) {

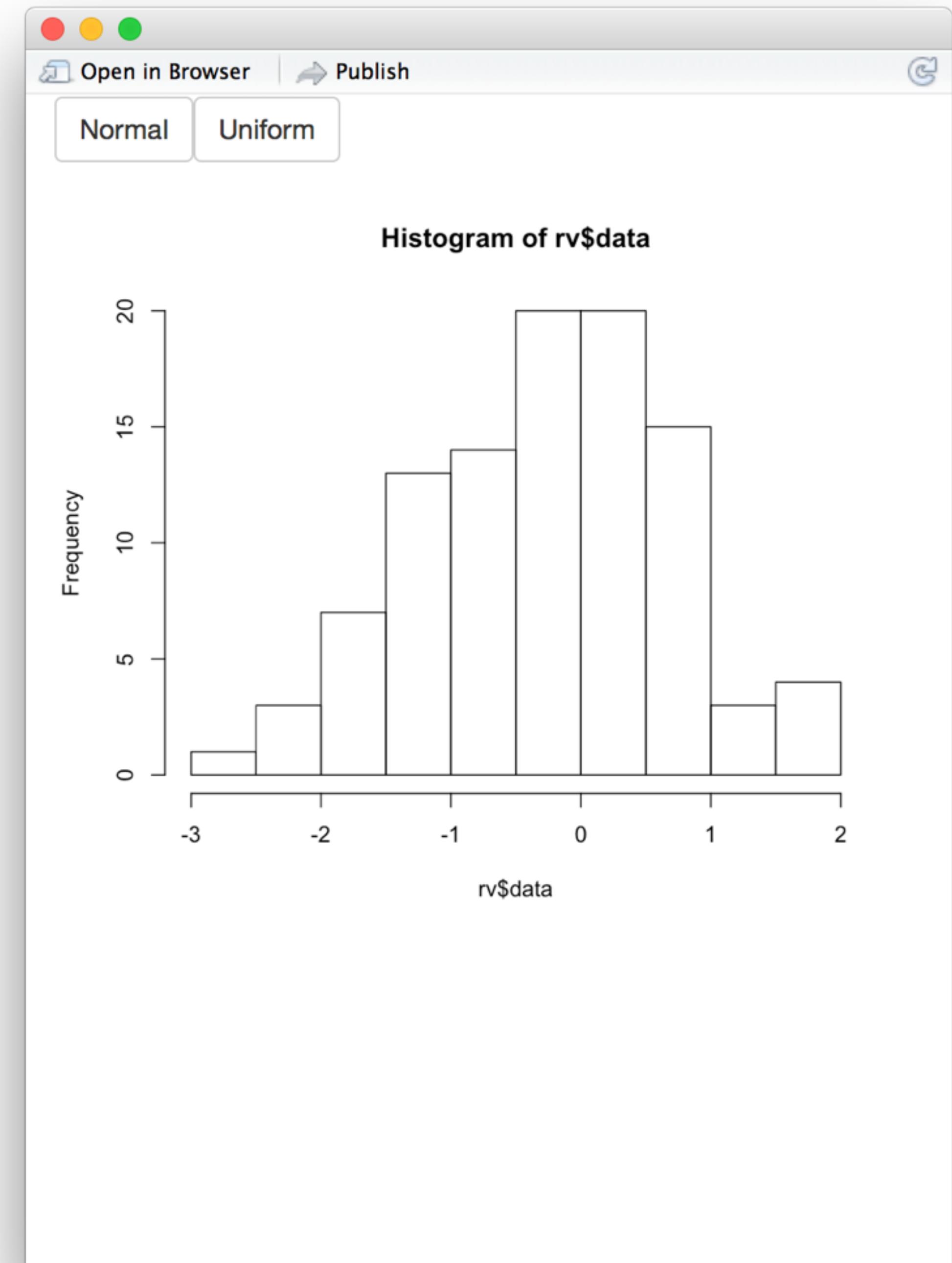
  rv <- reactiveValues(data = rnorm(100))

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

  output$hist <- renderPlot({
    hist(rv$data)
  })
}

shinyApp(ui = ui, server = server)

```



```

# 08-reactiveValues

library(shiny)

ui <- fluidPage(
  actionButton(inputId = "norm", label = "Normal"),
  actionButton(inputId = "unif", label = "Uniform"),
  plotOutput("hist")
)

server <- function(input, output) {

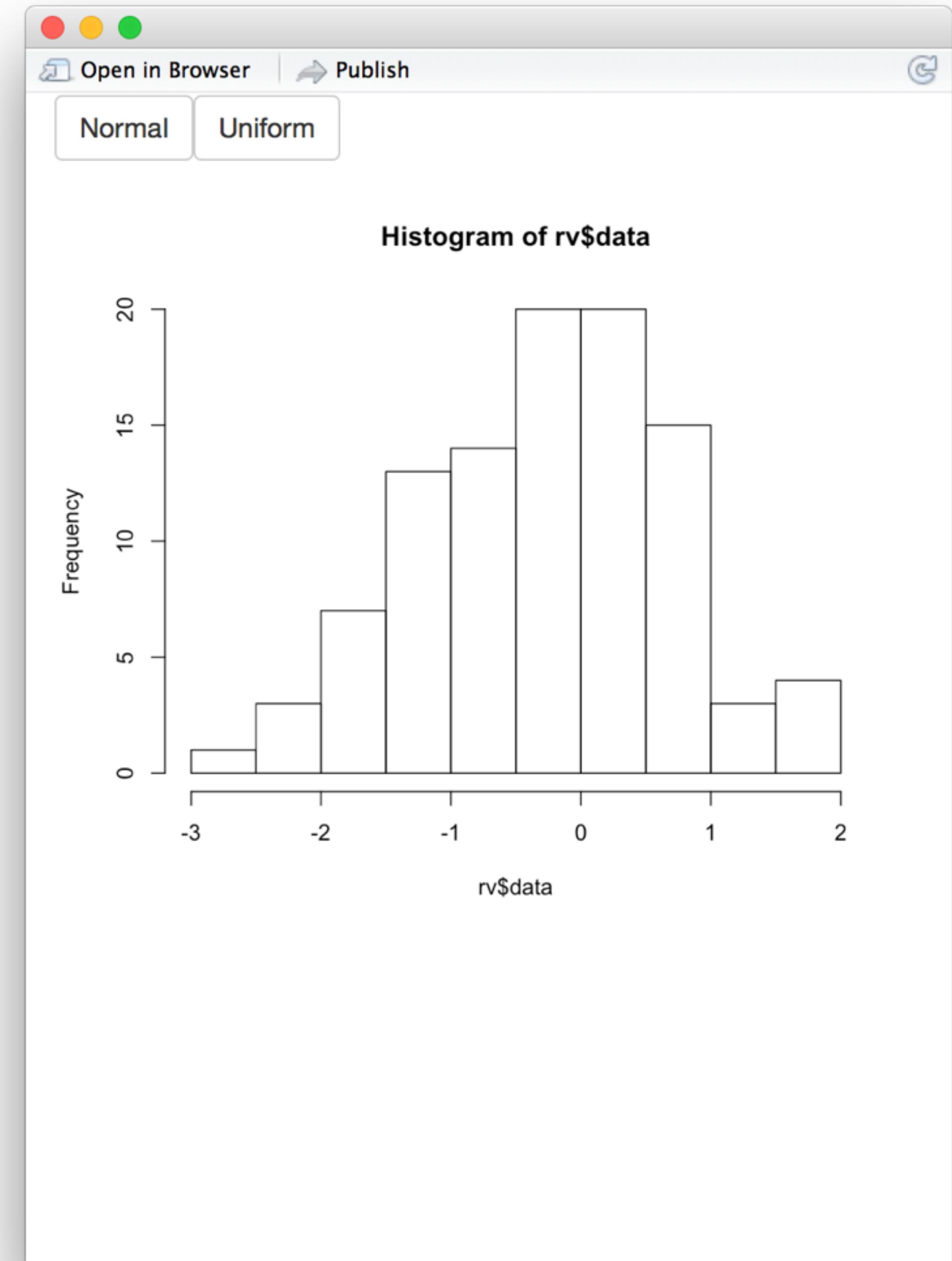
  rv <- reactiveValues(data = rnorm(100))

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

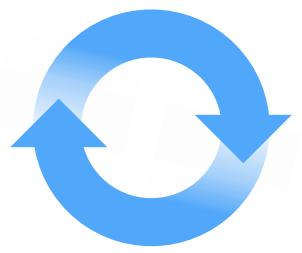
  output$hist <- renderPlot({
    hist(rv$data)
  })
}

shinyApp(ui = ui, server = server)

```



Recap: reactiveValues()

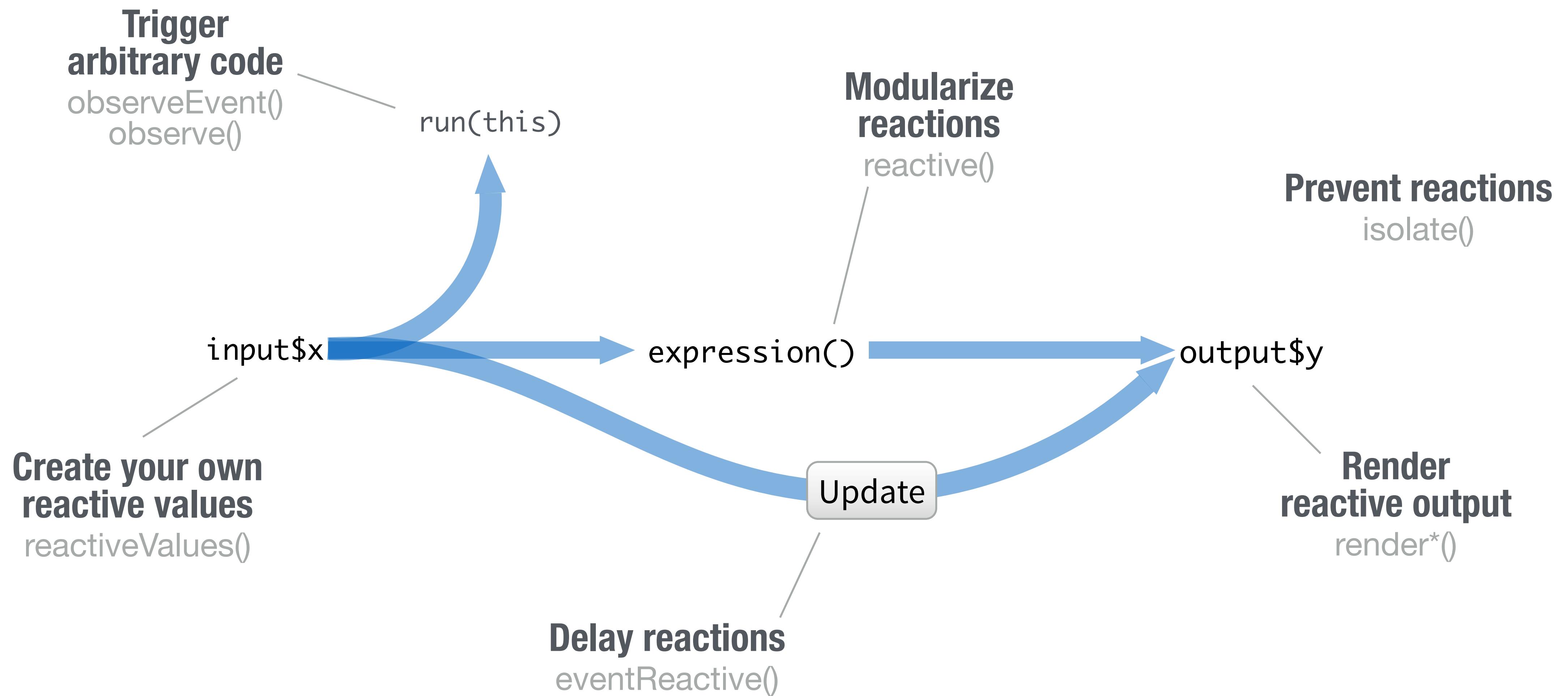


reactiveValues() creates a list of **reactive values**

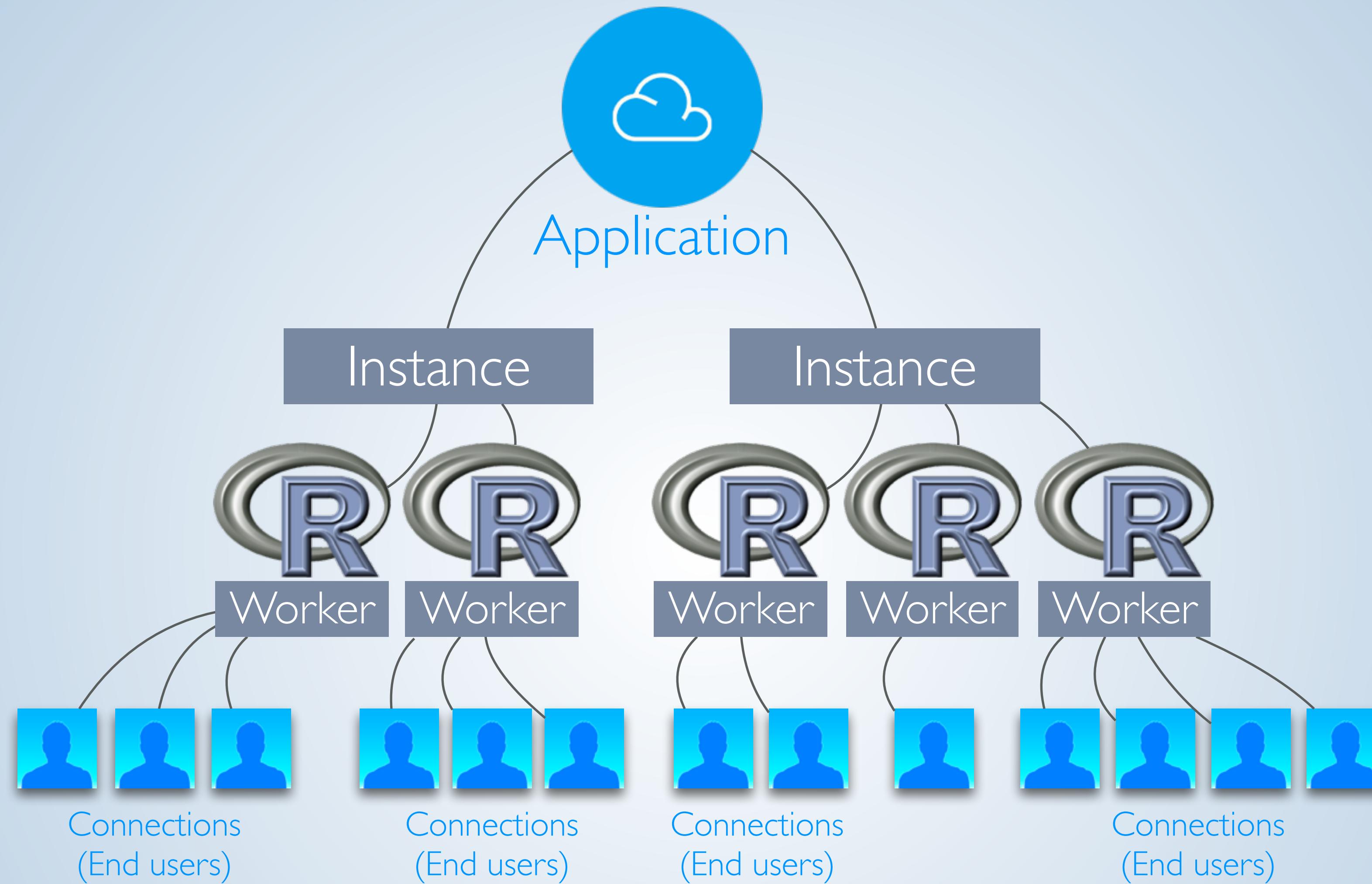
`rv$data <-`

You can manipulate these values (usually with observeEvent())

You now how to



Parting tips



Reduce repetition

Place code where it will be re-run as little as necessary

```
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num", label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)

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

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

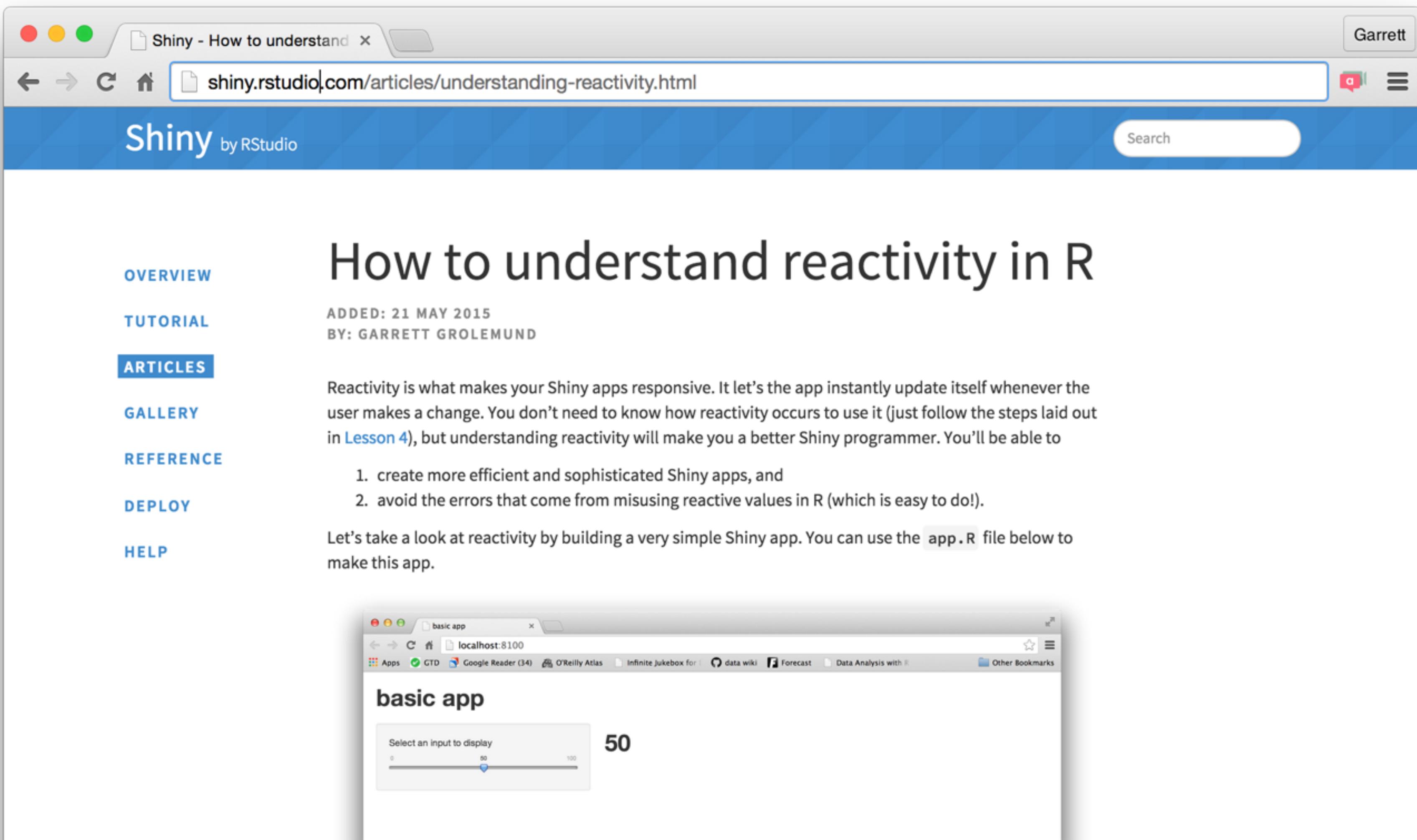
Code outside the server function will
be run once per R session (worker)

Code inside the server function will be
run once per end user (connection)

Code inside a reactive function will be
run once per reaction
(e.g. many times)

How can R possibly implement reactivity?

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



The screenshot shows a web browser window with the title bar "Shiny - How to understand" and the address bar containing the URL "shiny.rstudio.com/articles/understanding-reactivity.html". The browser interface includes standard controls like back, forward, and search. The main content area features a blue header with the "Shiny by RStudio" logo and a search bar. The main heading is "How to understand reactivity in R". To the left is a sidebar with navigation links: OVERVIEW, TUTORIAL, ARTICLES (which is selected), GALLERY, REFERENCE, DEPLOY, and HELP. Below the sidebar, the article details are listed: "ADDED: 21 MAY 2015" and "BY: GARRETT GROLEMUND". The article text explains what reactivity is and how it makes Shiny apps responsive. It also lists two goals for understanding reactivity: creating more efficient apps and avoiding errors from misusing reactive values. At the bottom, there's a note about building a simple Shiny app using an `app.R` file. A smaller inset screenshot at the bottom shows a "basic app" window with a slider control set to 50.

Garrett

shiny.rstudio.com/articles/understanding-reactivity.html

Shiny by RStudio

Search

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

How to understand reactivity in R

ADDED: 21 MAY 2015
BY: GARRETT GROLEMUND

Reactivity is what makes your Shiny apps responsive. It lets the app instantly update itself whenever the user makes a change. You don't need to know how reactivity occurs to use it (just follow the steps laid out in [Lesson 4](#)), but understanding reactivity will make you a better Shiny programmer. You'll be able to

1. create more efficient and sophisticated Shiny apps, and
2. avoid the errors that come from misusing reactive values in R (which is easy to do!).

Let's take a look at reactivity by building a very simple Shiny app. You can use the `app.R` file below to make this app.

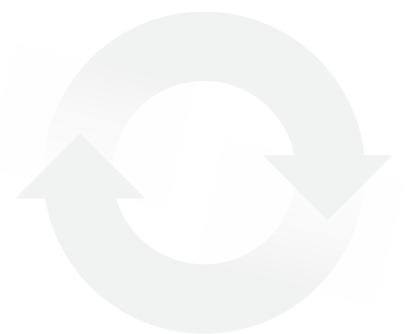
basic app

Select an input to display

50

Learn
more

How to start with Shiny



1. How to build a Shiny app (www.rstudio.com/resources/webinars/)
2. How to customize reactions (Today)
3. How to customize appearance (June 3)

The Shiny Development Center

shiny.rstudio.com

