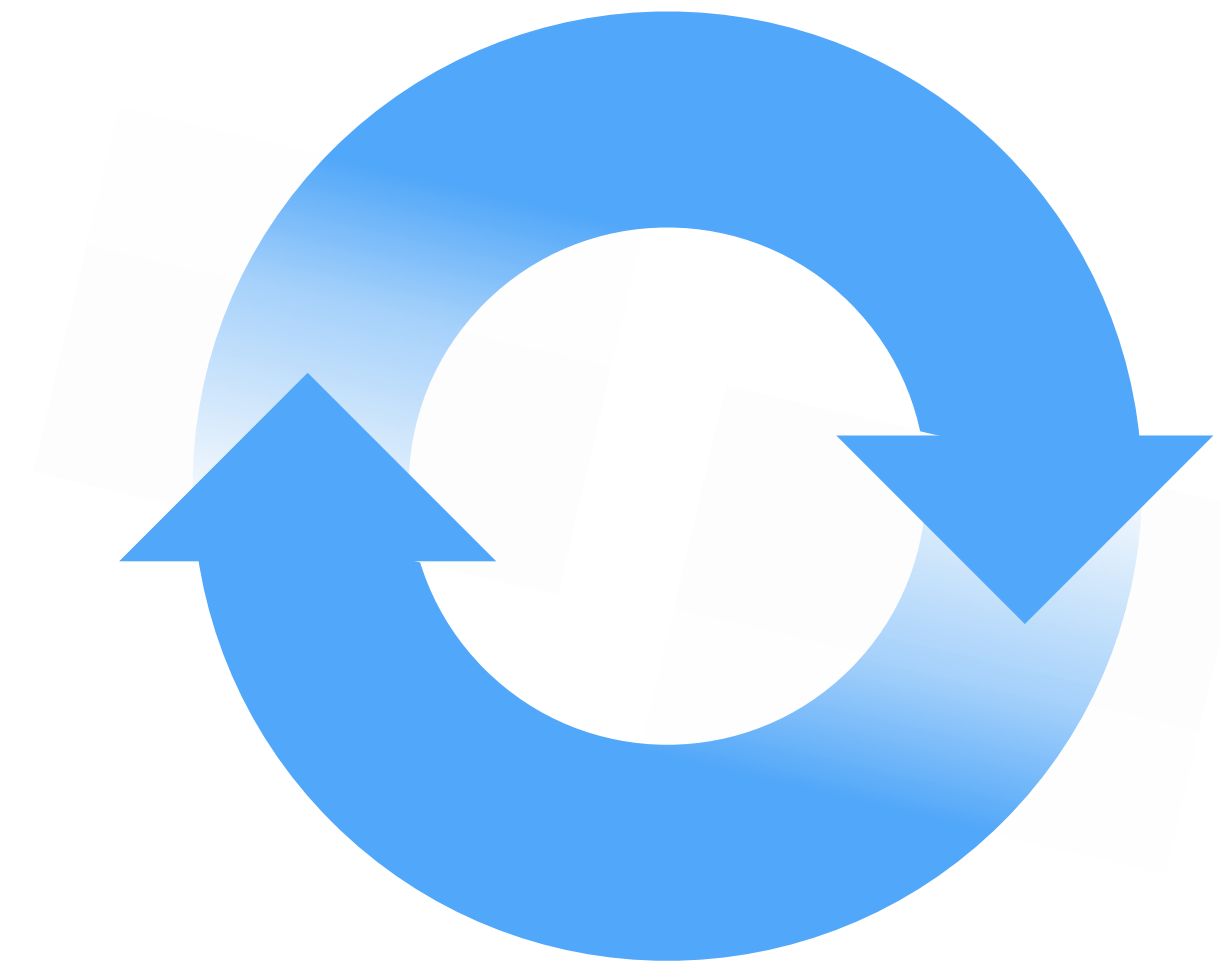


All Training materials are provided "as is" and without warranty and RStudio disclaims any and all express and implied warranties including without limitation the implied warranties of title, fitness for a particular purpose, merchantability and noninfringement.

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 Grolemond**

Data Scientist and Master Instructor

May 2015

Email: [garrett@rstudio.com](mailto:garrett@rstudio.com)

Twitter: [@StatGarrett](https://twitter.com/StatGarrett)

Code and slides at:  
[bit.ly/shiny-quickstart-2](http://bit.ly/shiny-quickstart-2)

### 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.

# Shiny Showcase

[www.rstudio.com/products/shiny/shiny-user-showcase/](http://www.rstudio.com/products/shiny/shiny-user-showcase/)

### 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.

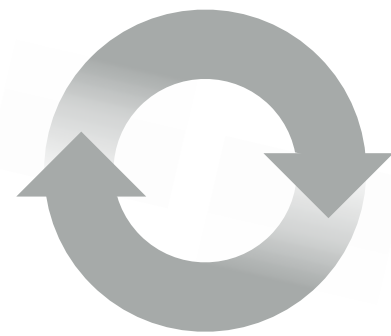
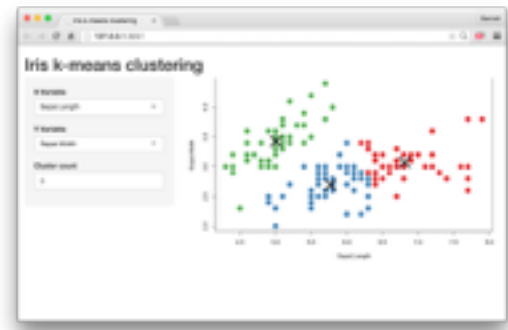


Ebola Model

An epidemiological simulation.



# How to start with Shiny



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



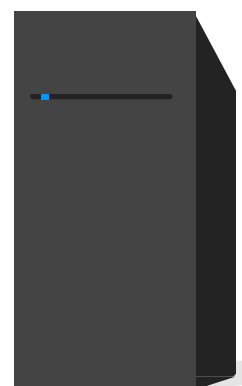
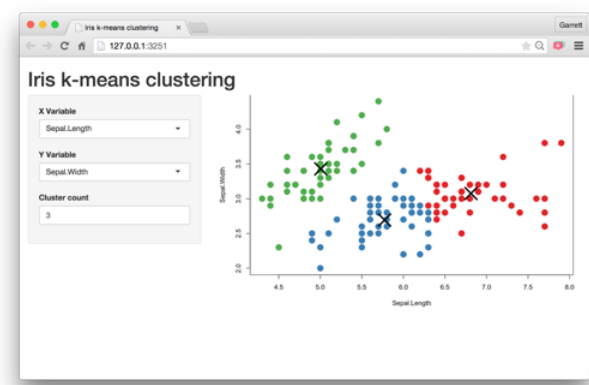
**The story**  
**so far**

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)

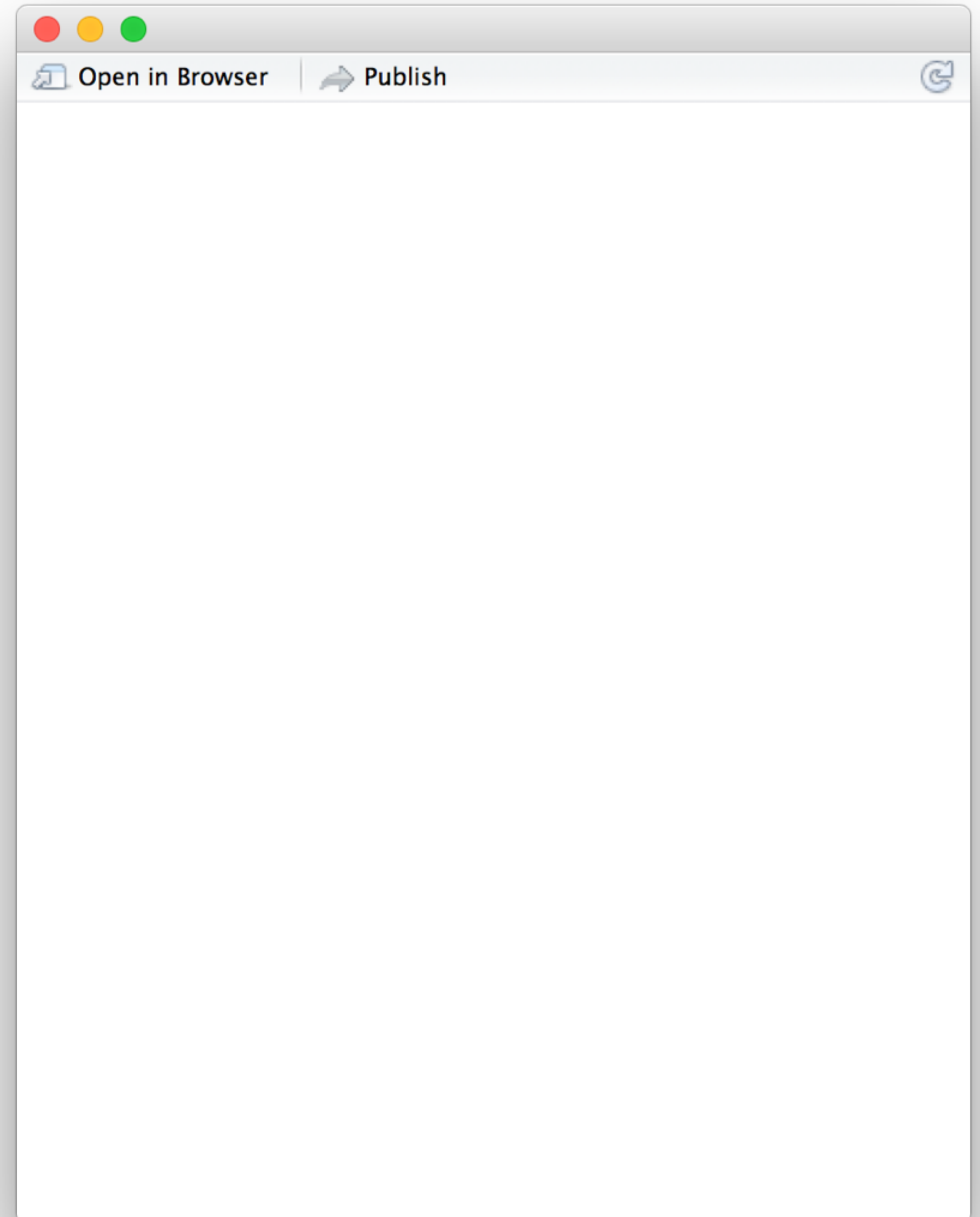
ui <- fluidPage(

)

server <- function(input, output) {

}

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



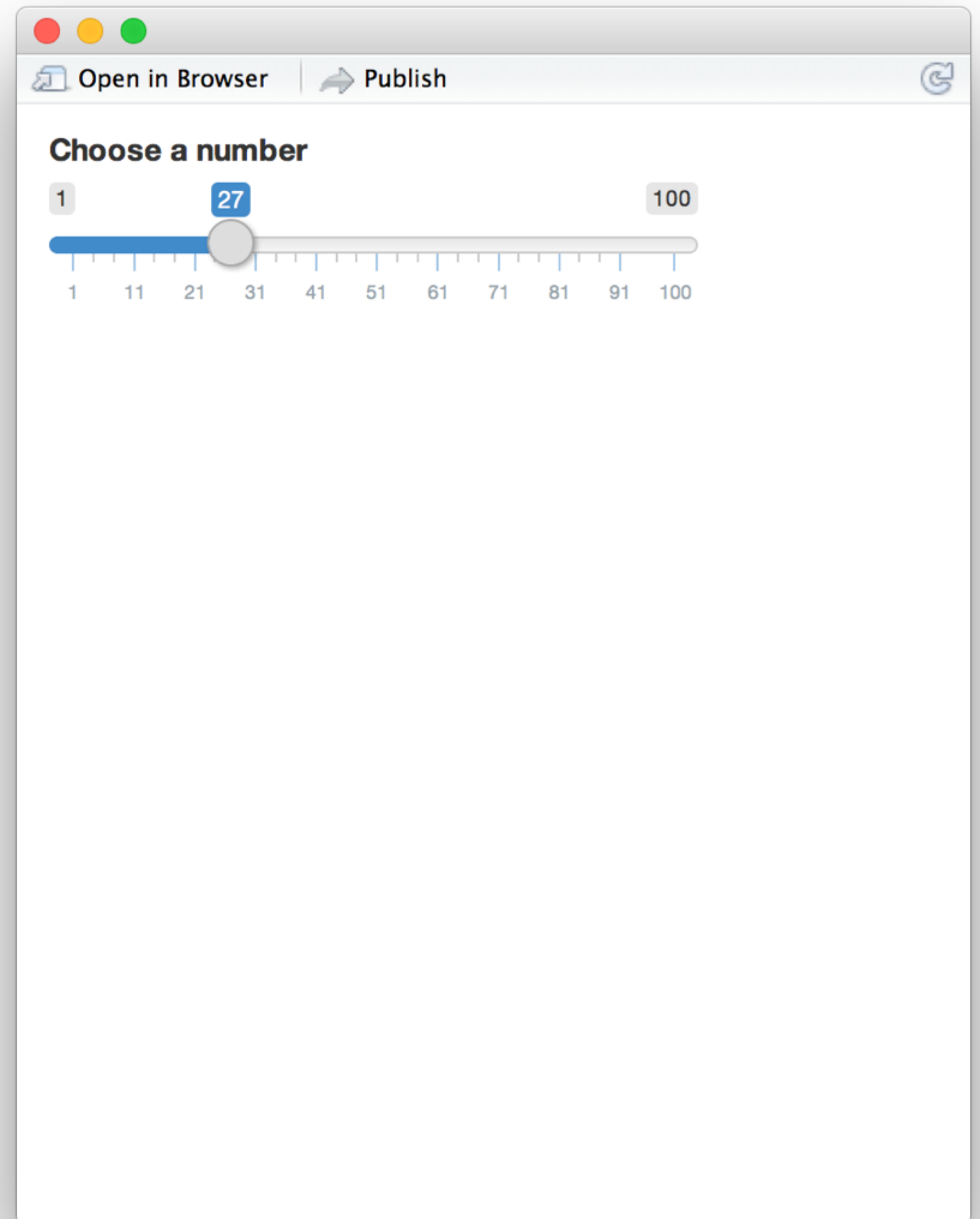
```
library(shiny)

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)
```



The screenshot shows a Shiny web application window. At the top, there are three colored window control buttons (red, yellow, green) and two buttons: "Open in Browser" and "Publish". The main content area has the title "Choose a number" and a slider input. The slider has a range from 1 to 100, with major tick marks every 10 units. The current value is 27, indicated by a blue box above the slider handle. Below the slider, there is a plot output area, which is currently empty. A text box with a black border is overlaid on the bottom half of the plot area, containing the following text:

- \* Output() adds a space in the ui for an R object.

**You must build the object in the server function**

```
library(shiny)

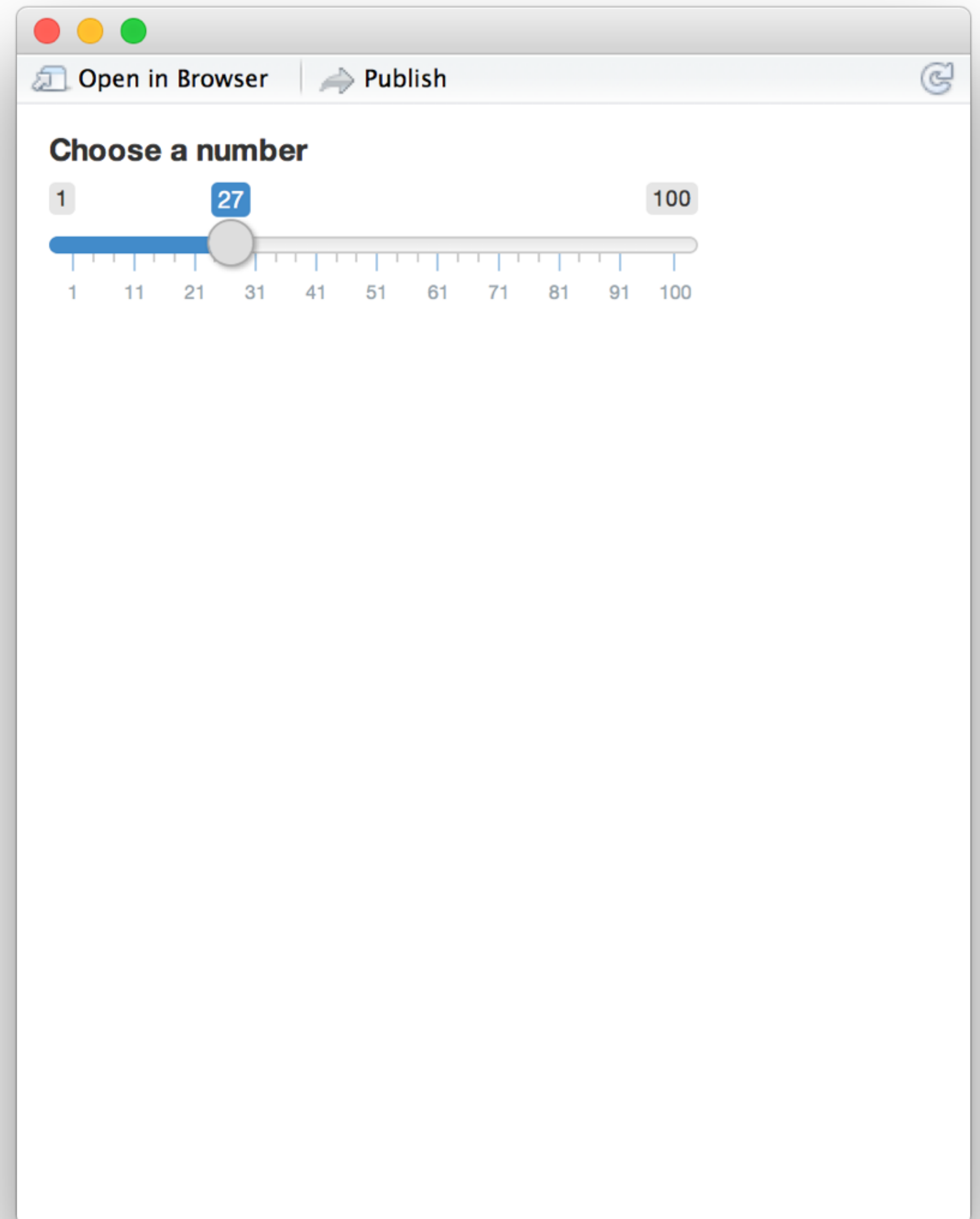
ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)

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

}

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

# 1



```
library(shiny)

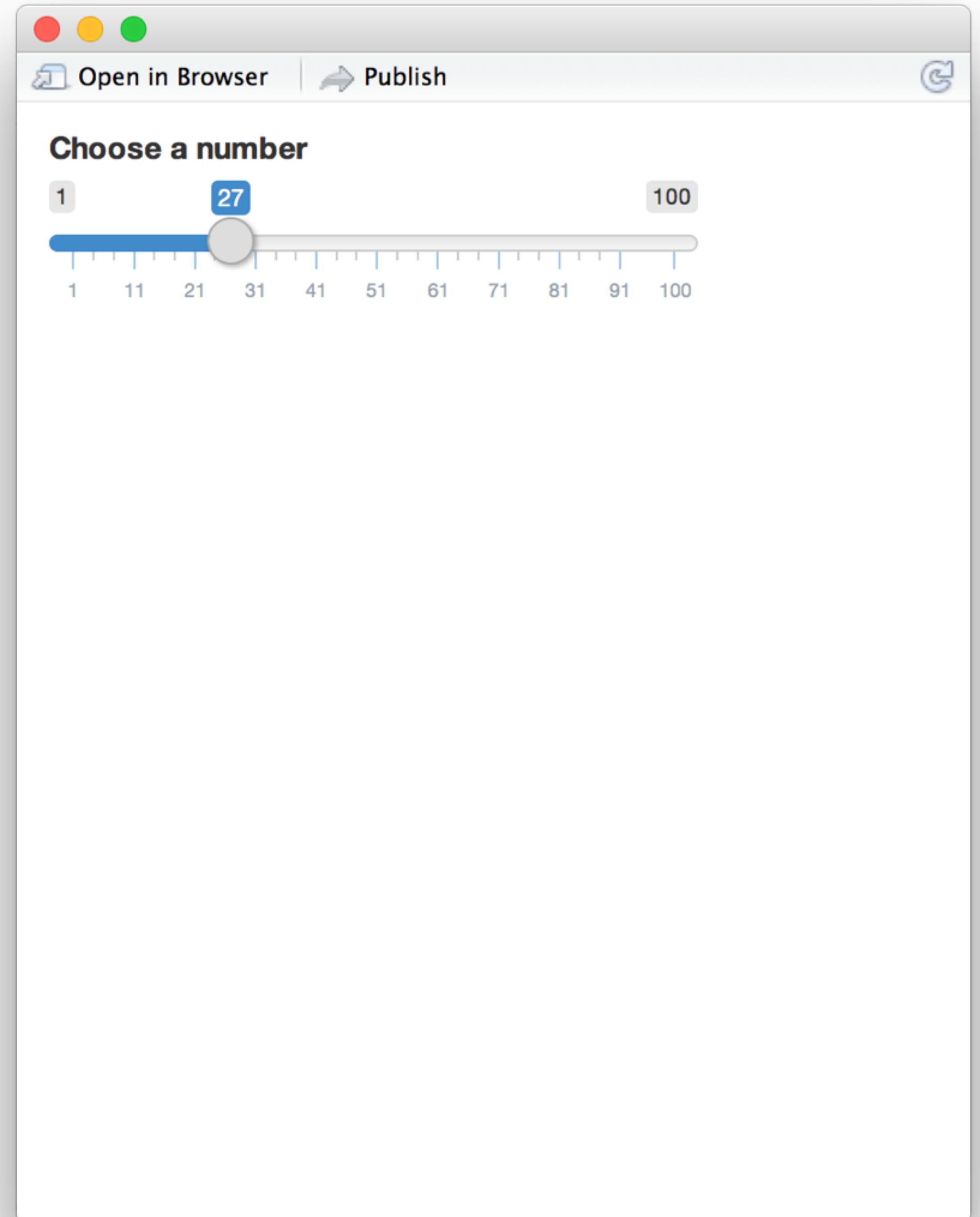
ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)

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

}}
}

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

2



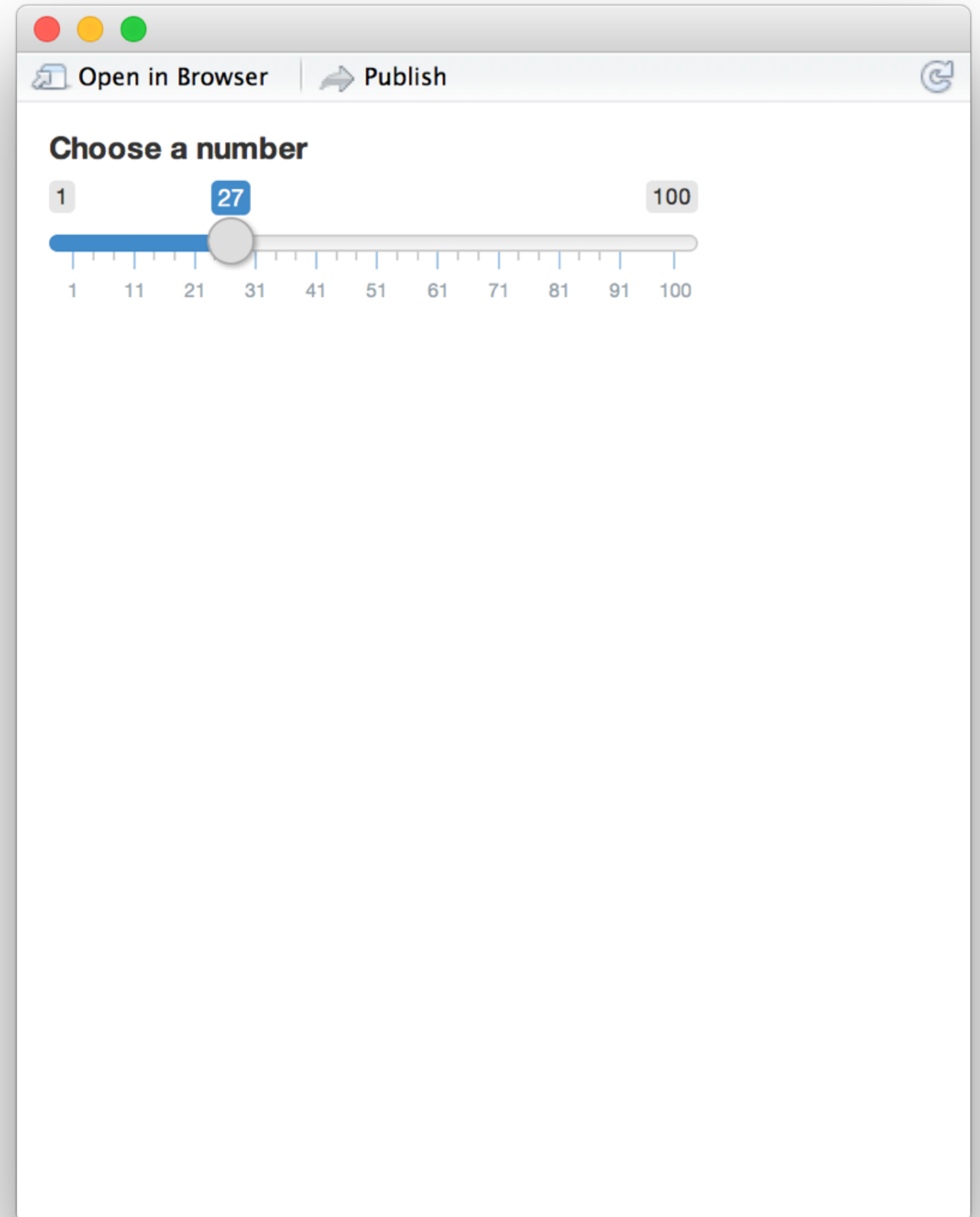
```
library(shiny)

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



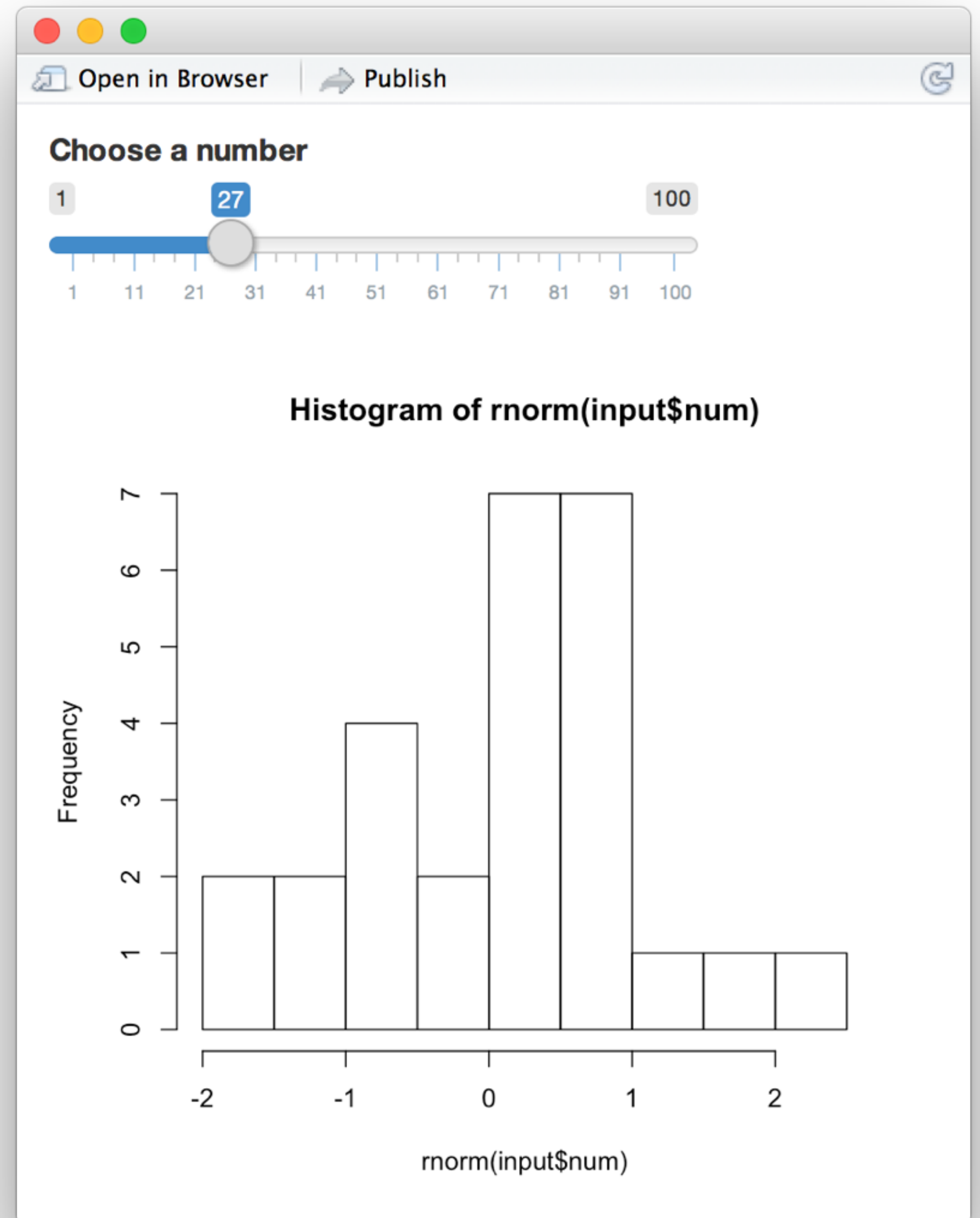


```
library(shiny)

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

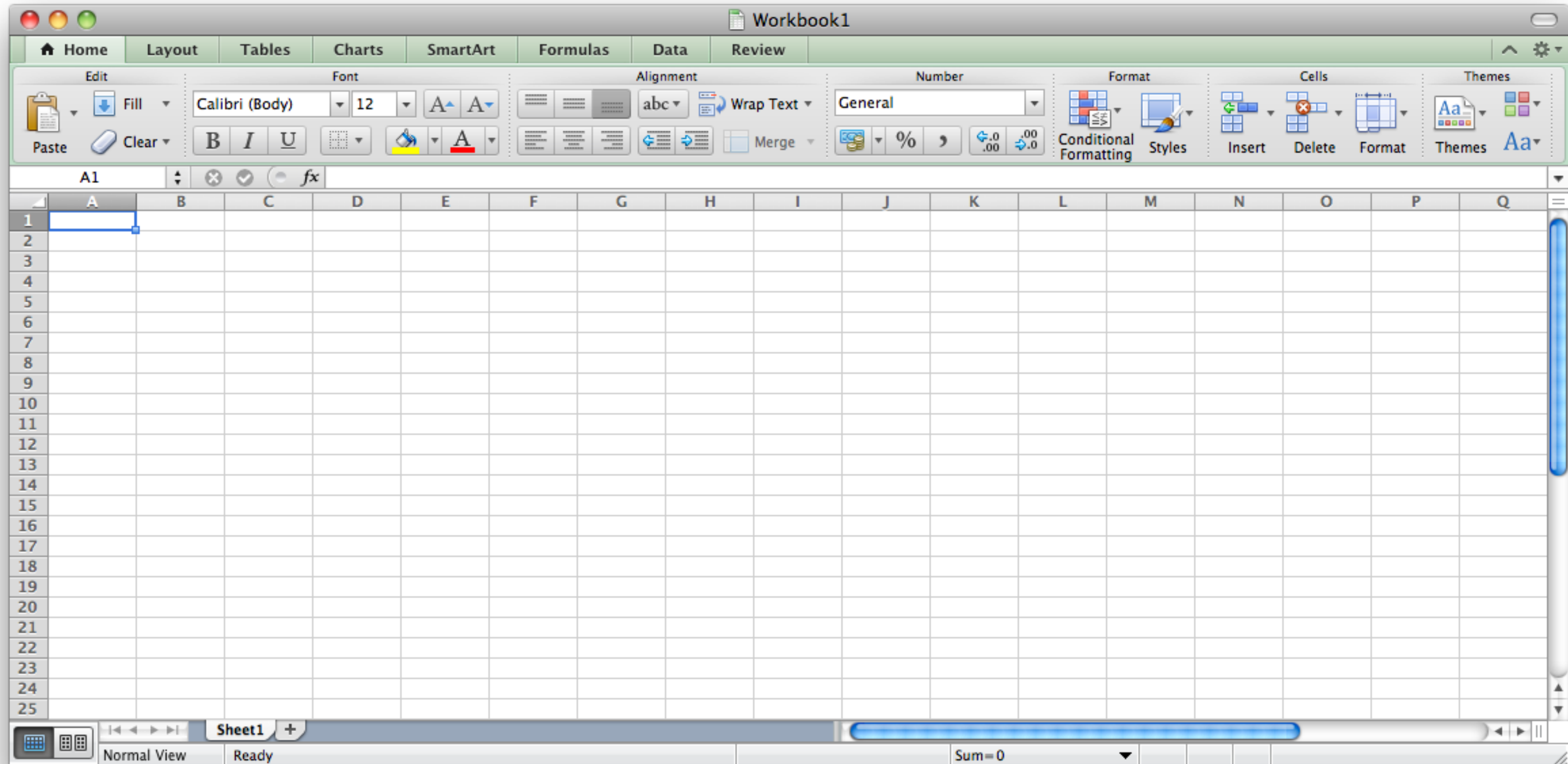


**Shiny Server (Pro)**

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

**What is  
Reactivity?**

# Think Excel.



Workbook1

Tables | Charts | SmartArt | Formulas | Data | Review

Font: Arial (Body), 12, Bold, Italic, Underline, Text Color, Background Color

Alignment: Left, Center, Right, Wrap Text, Merge

Number: General, Percentage, Decimals

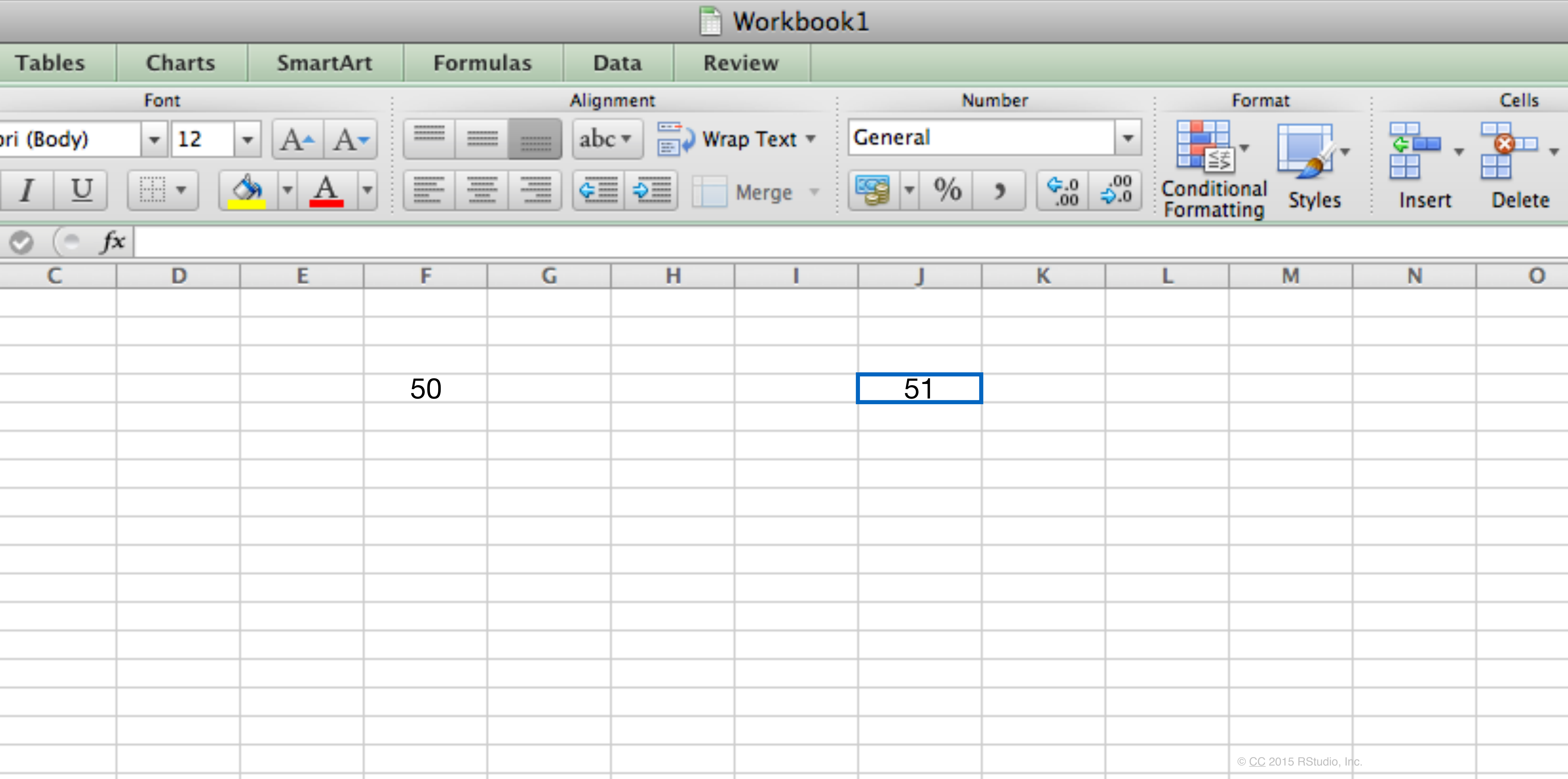
Format: Conditional Formatting, Styles

Cells: Insert, Delete

C	D	E	F	G	H	I	J	K	L	M	N	O
			50				= F4 + 1					

fx

© CC 2015 RStudio, Inc.







Workbook1

Tables | Charts | SmartArt | Formulas | Data | Review

Font: Arial (Body), 12, Bold, Underline, Italic, Text Color, Background Color

Alignment: Left, Center, Right, Wrap Text, Merge

Number: General, Percentage, Decimals

Format: Conditional Formatting, Styles

Cells: Insert, Delete

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: Arial (Body), 12, Bold, Italic, Underline, Text Color, Background Color

Alignment: Left, Center, Right, Wrap Text, Merge

Number: General, Currency, Percentage, Increase/Decrease Decimal

Format: Conditional Formatting, Styles

Cells: Insert, Delete

C	D	E	F	G	H	I	J	K	L	M	N	O
			1000				1001					

© CC 2015 RStudio, Inc.

Workbook1

Tables | Charts | SmartArt | Formulas | Data | Review

Font: Arial (Body), 12, Bold, Italic, Underline, Text Color, Background Color

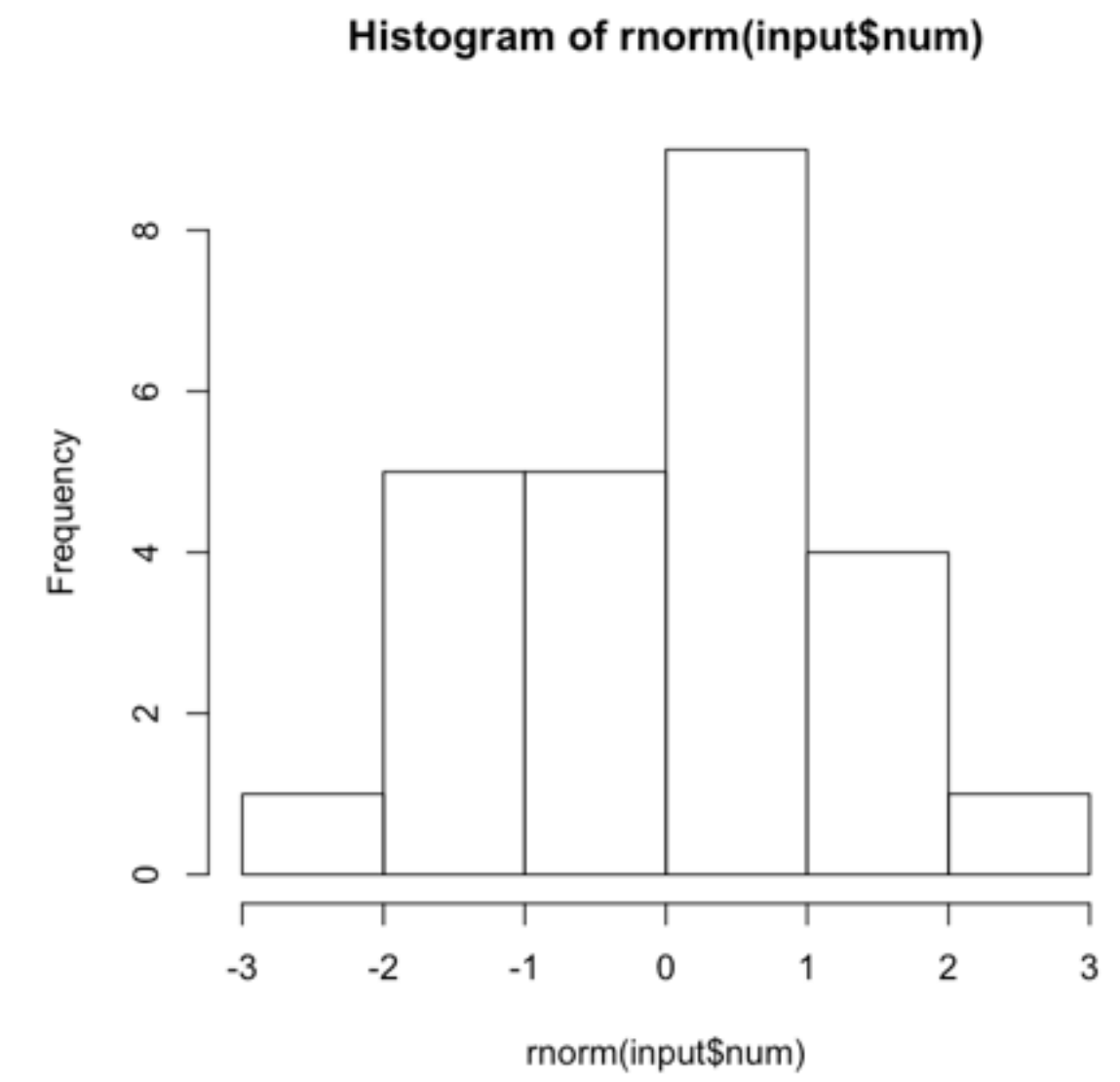
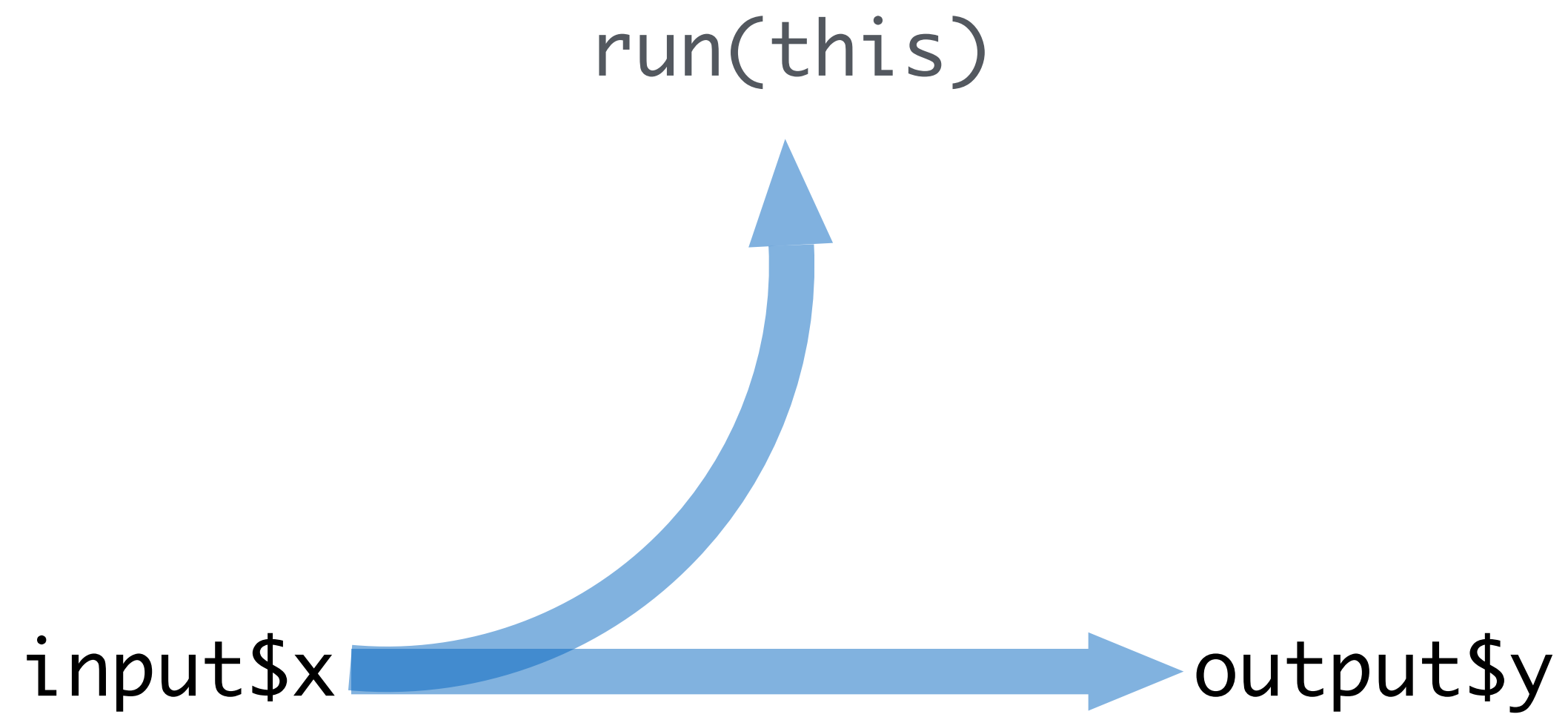
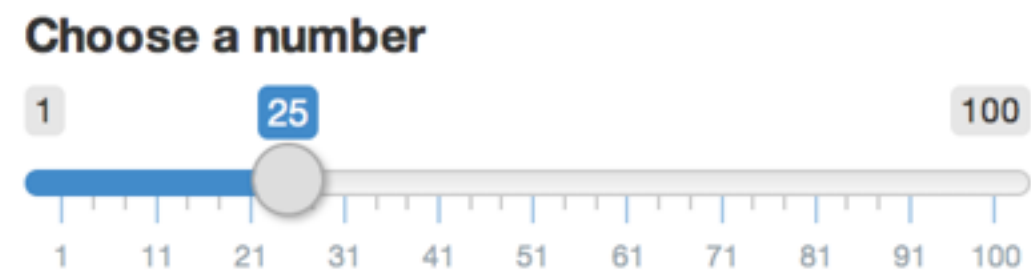
Alignment: Left, Center, Right, Wrap Text, Merge

Number: General, Currency, Percentage, Thousand Separator, Increase/Decrease Decimal Places

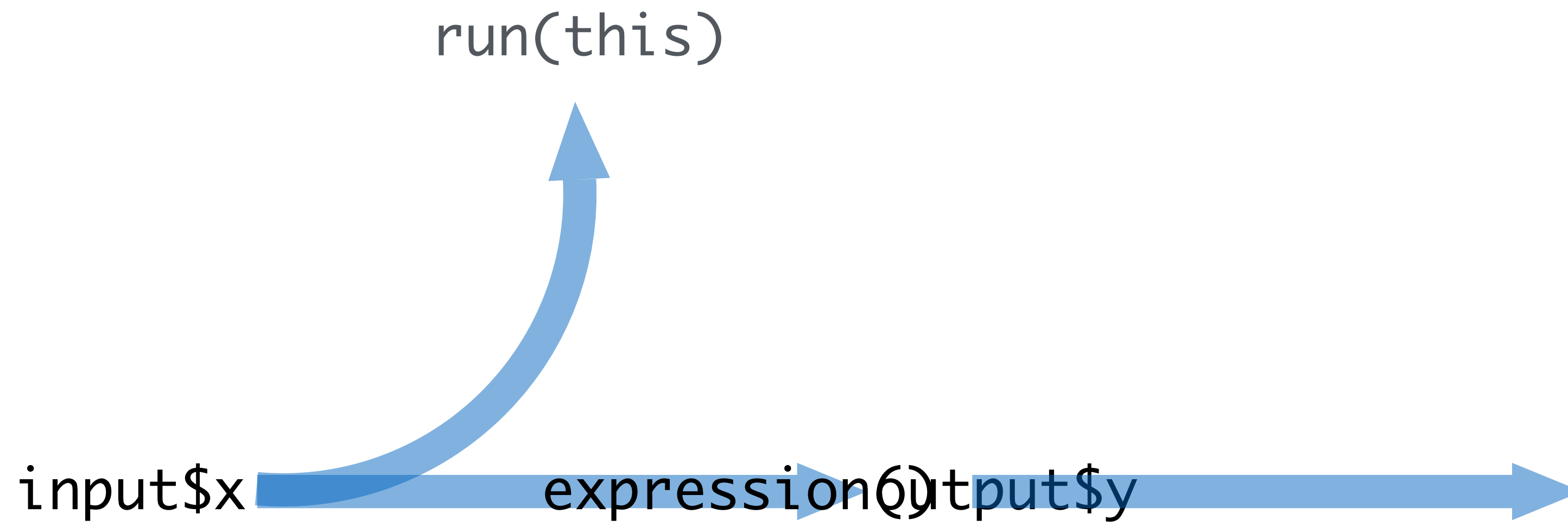
Format: Conditional Formatting, Styles

Cells: Insert, Delete

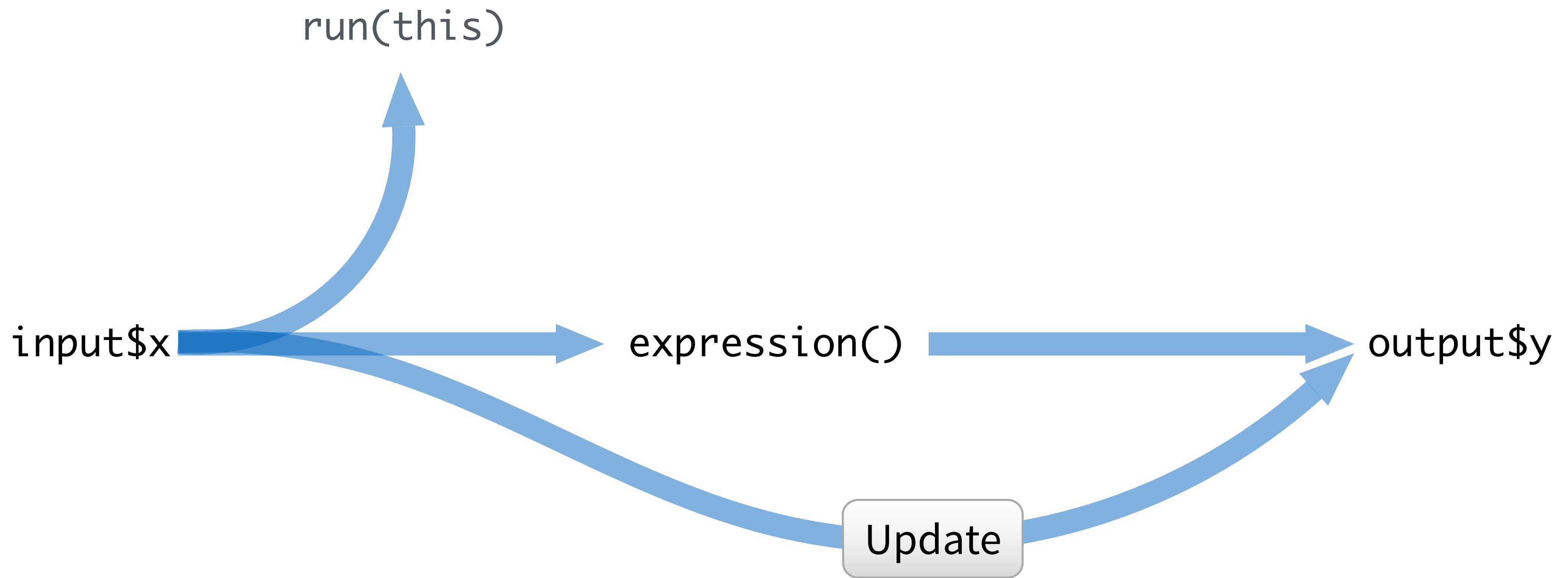
C	D	E	F	G	H	I	J	K	L	M	N	O
			1000					1001				
			input\$x					output\$y				







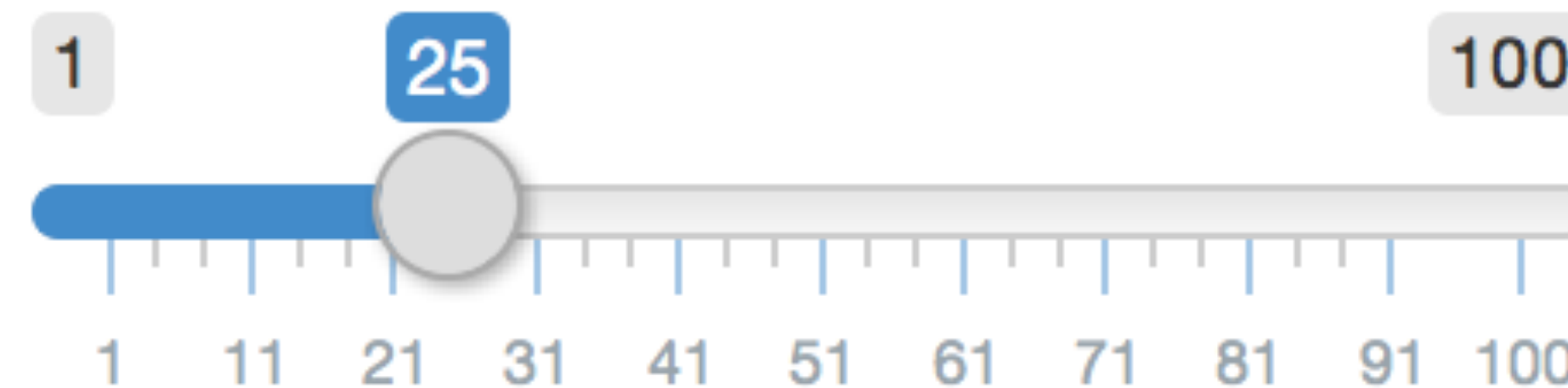




**Begin with**  
**reactive values**

# Syntax

Choose a number

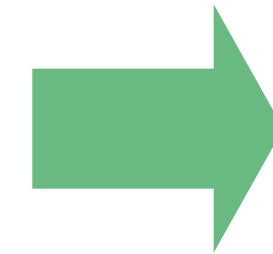
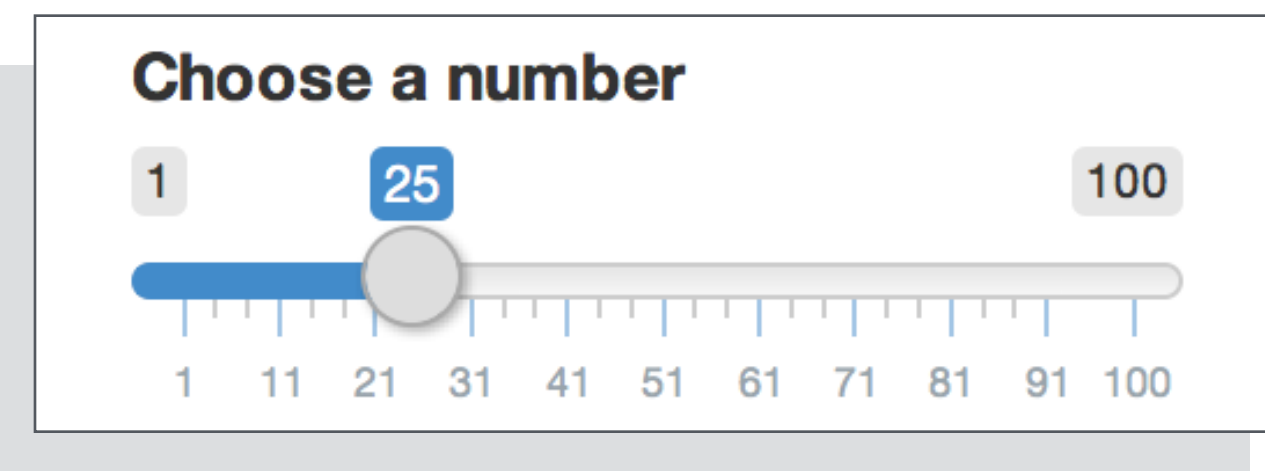


```
sliderInput(inputId = "num", label = "Choose a number", ...)
```

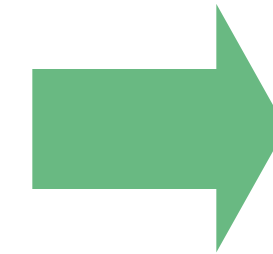
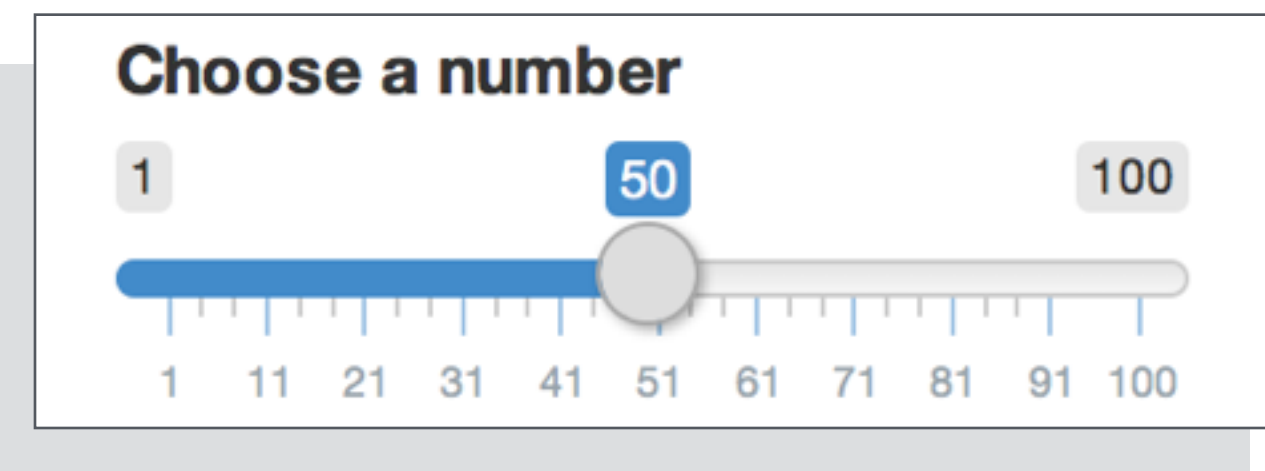
this input will provide a value  
saved as `input$num`

# Input values

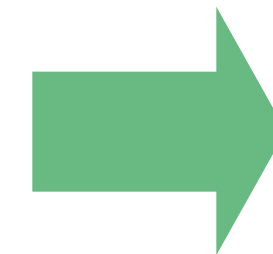
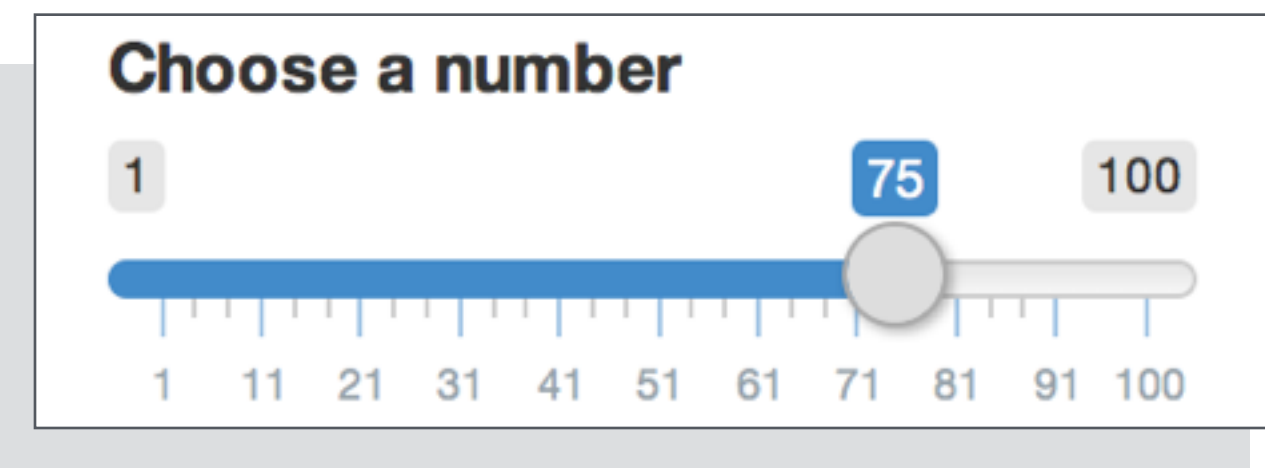
The input value changes whenever a user changes the input.



```
input$num = 25
```



```
input$num = 50
```



```
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))
```

```

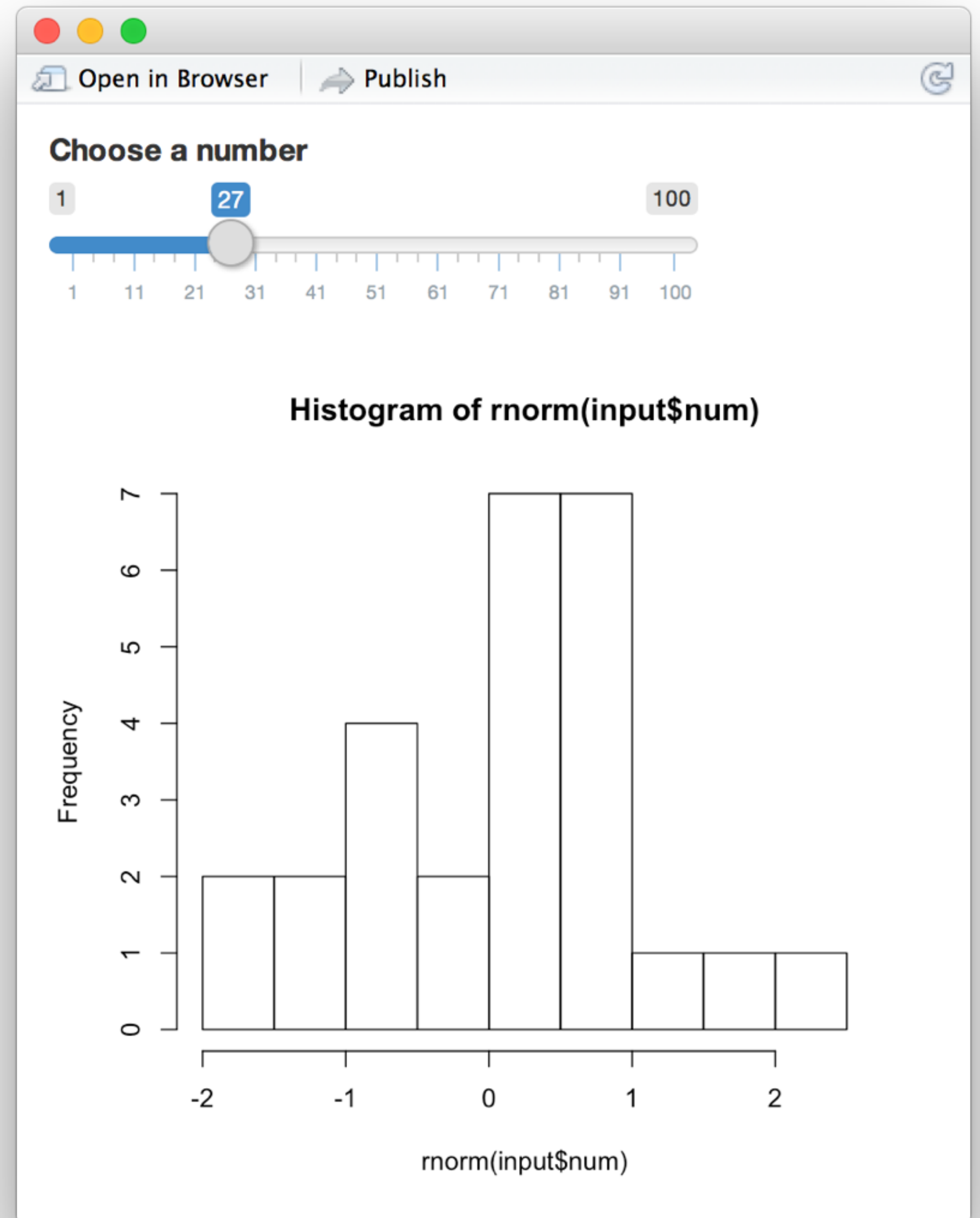
library(shiny)

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)

```



```

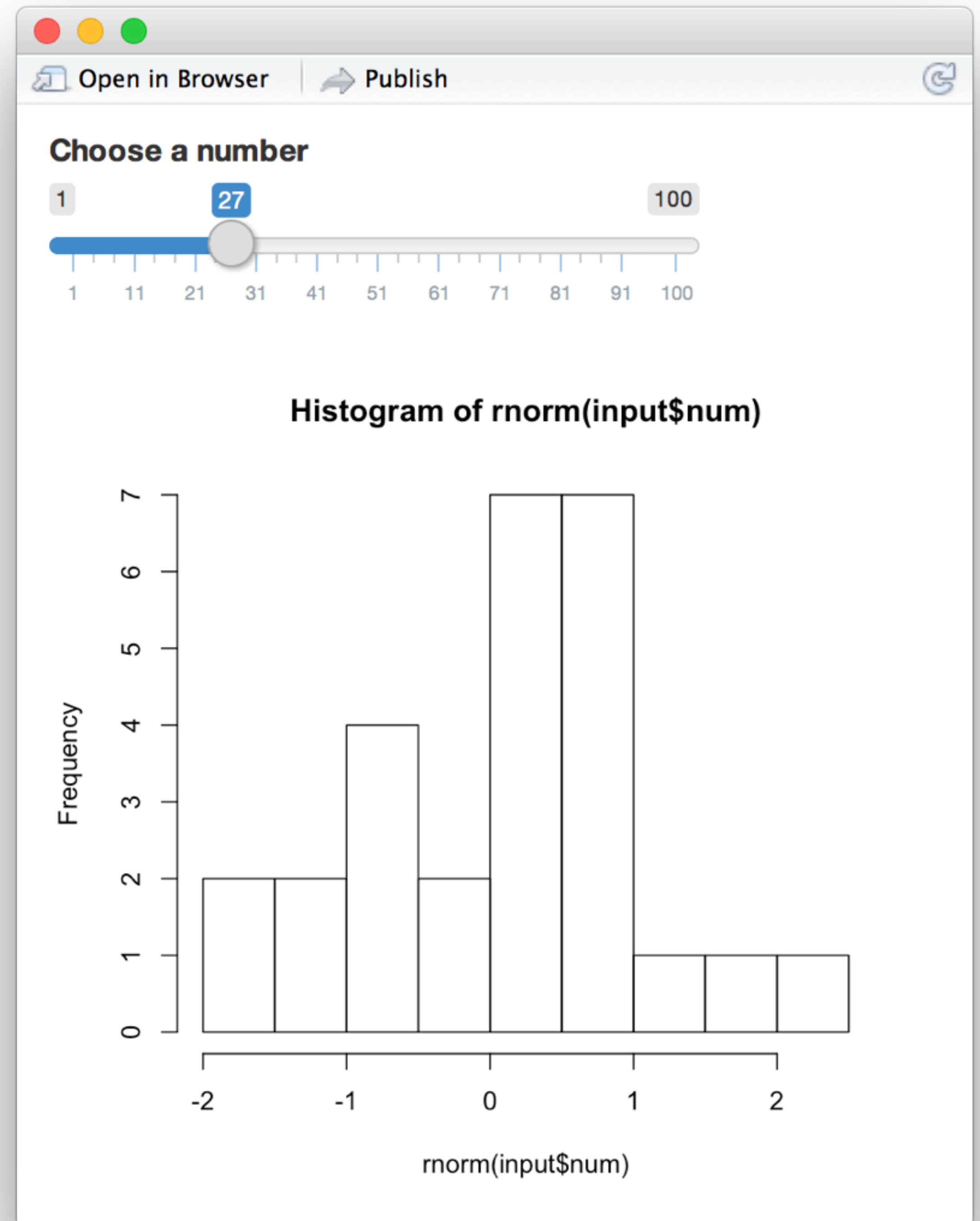
library(shiny)

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))
}

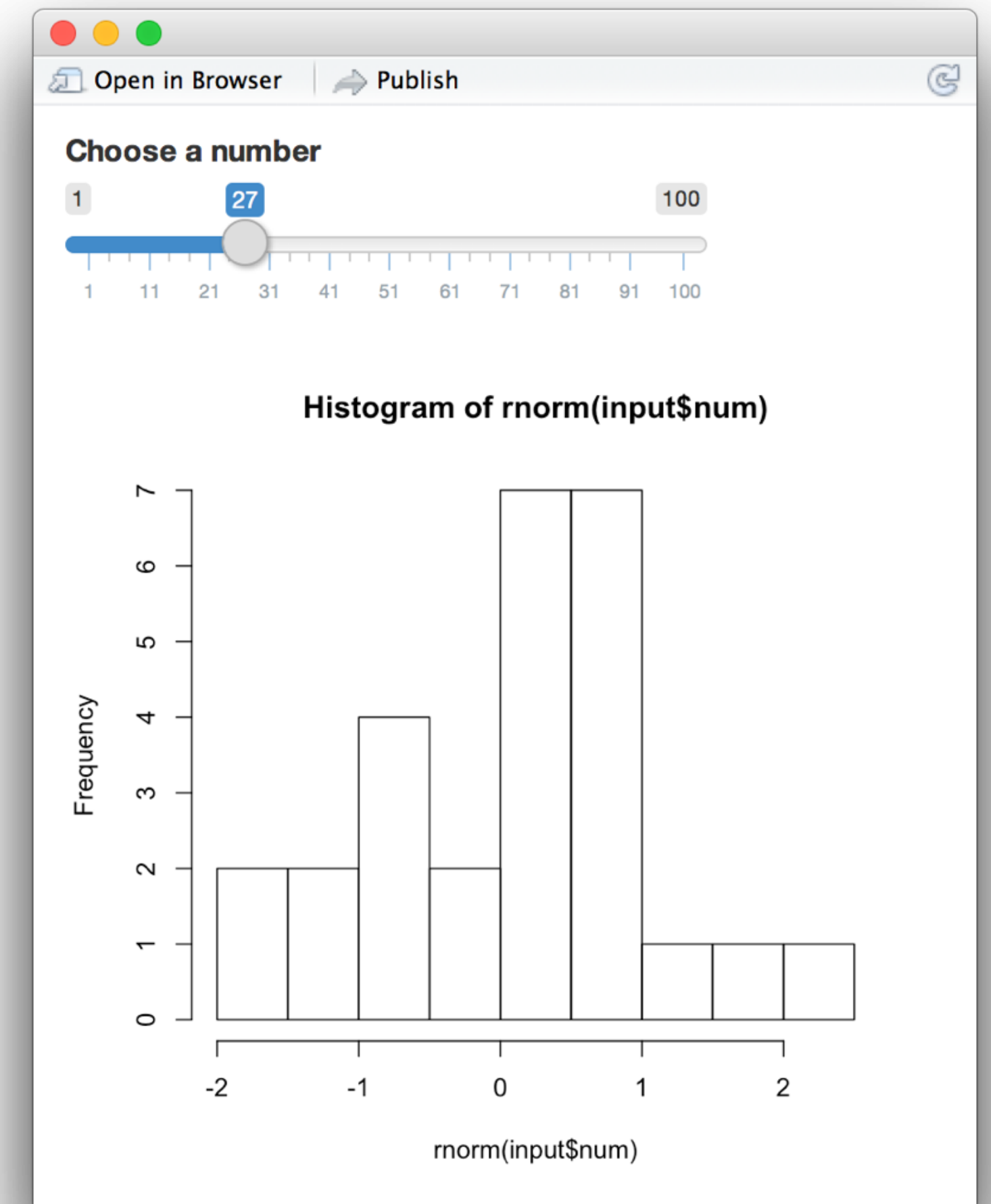
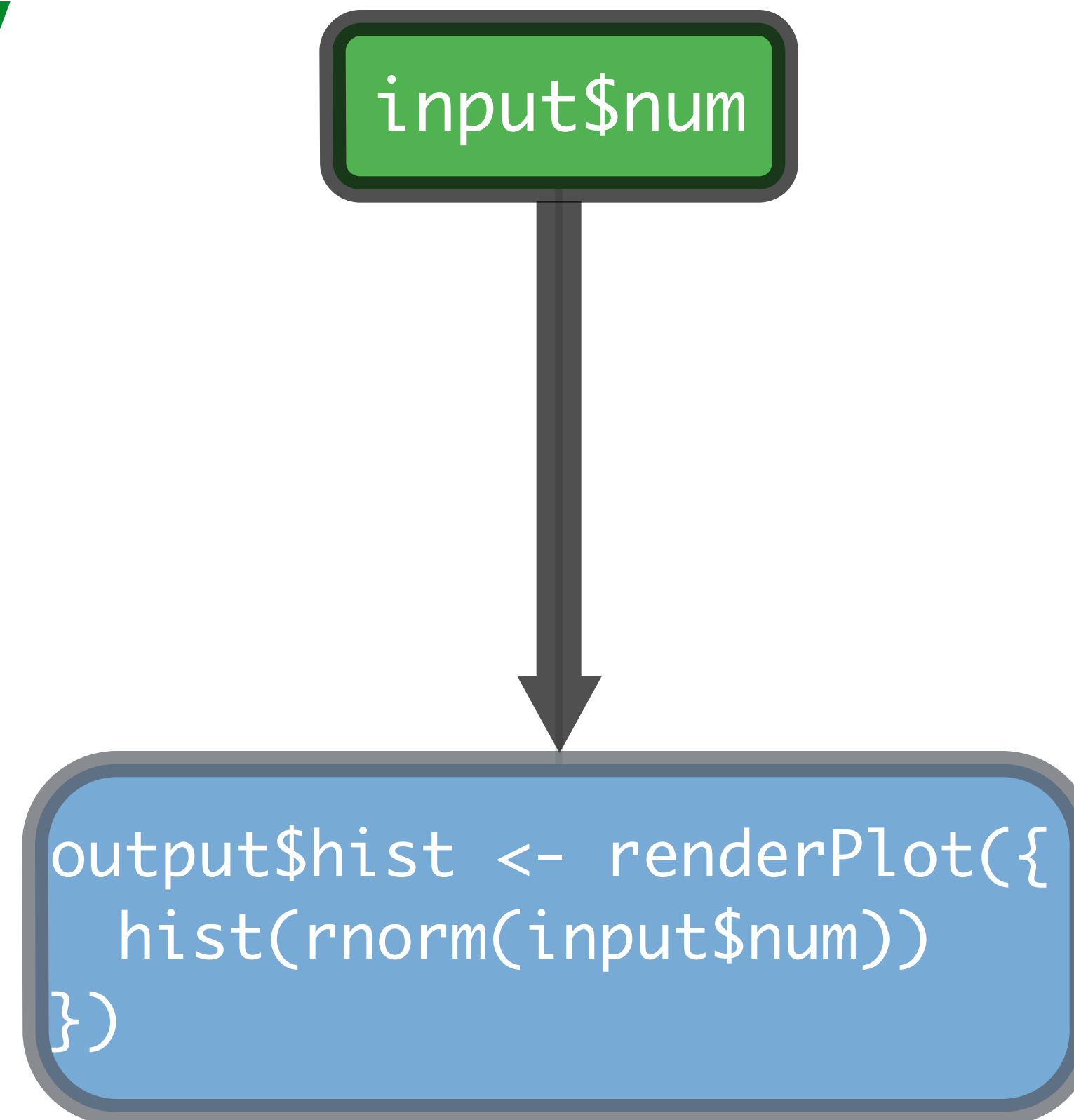
shinyApp(ui = ui, server = server)
```

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

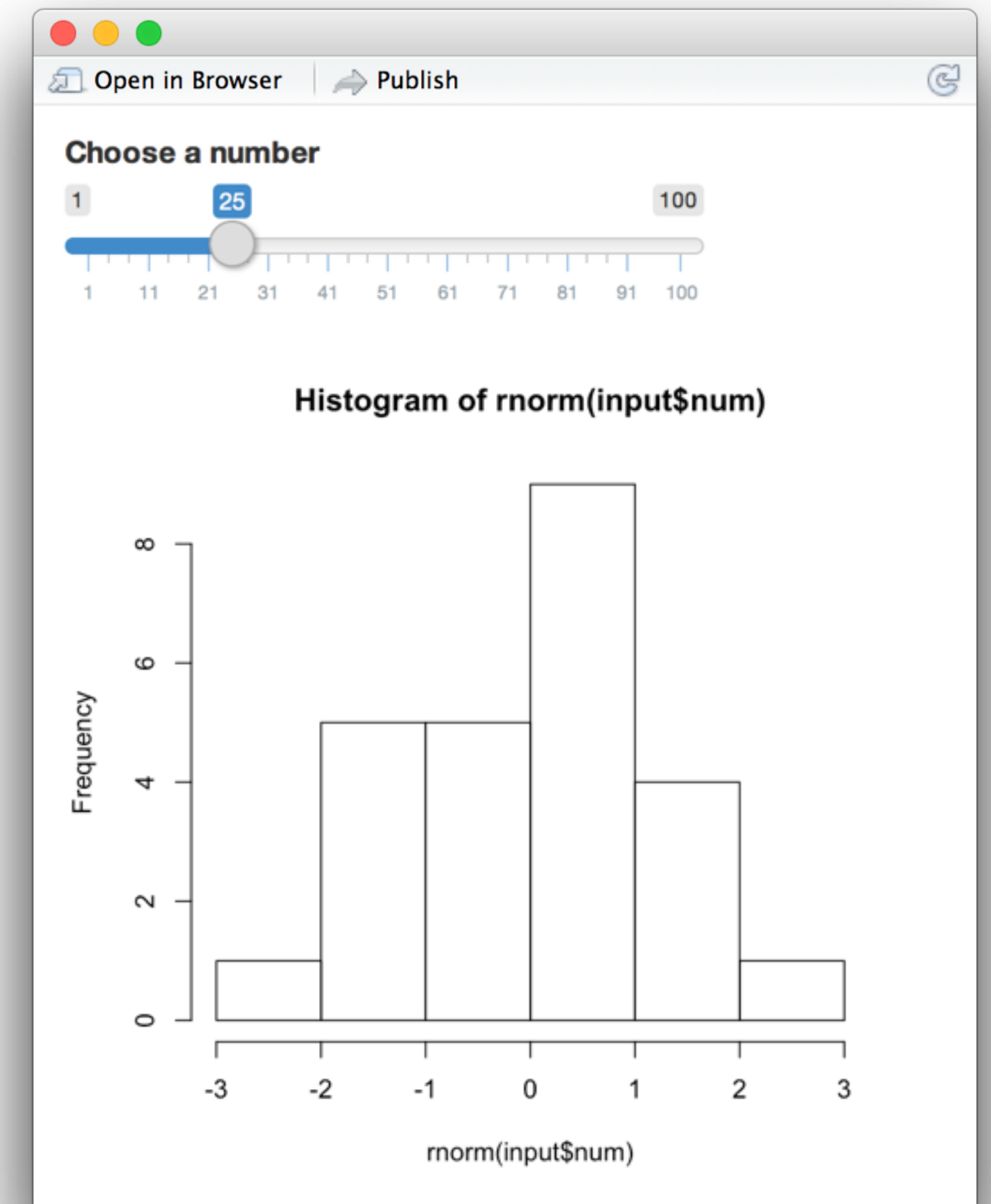
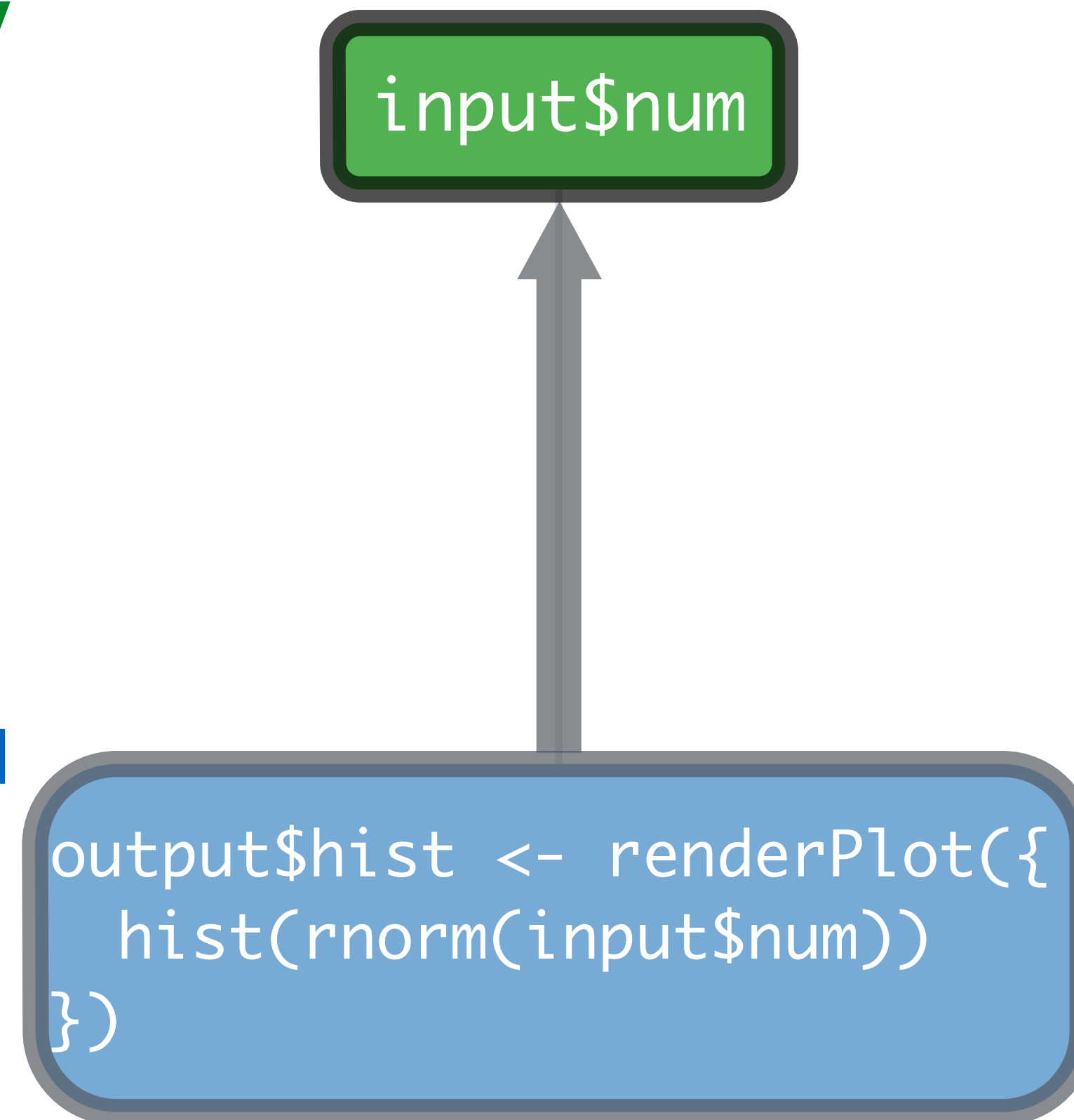
- 1 Reactive values notify**  
the functions that use them  
when they become invalid



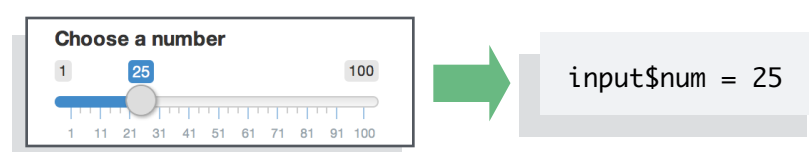
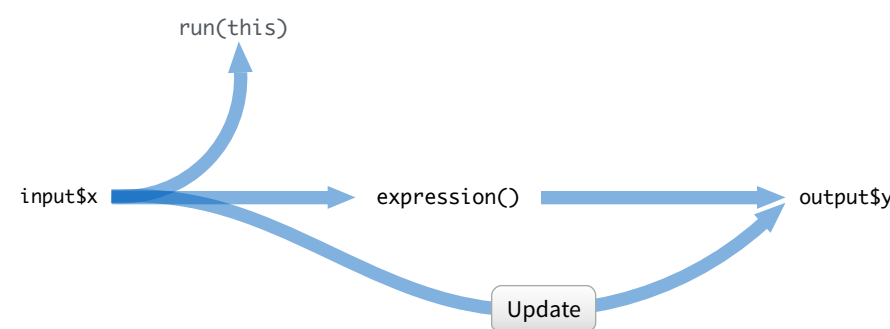
# Think of reactivity in R as a two step process

**1** **Reactive values notify**  
the functions that use them  
when they become invalid

**2** The objects created by  
**reactive functions respond**  
(different objects respond differently)



# Recap: Reactive values

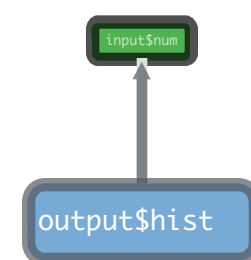


Reactive values act as the data streams that flow through your app.

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

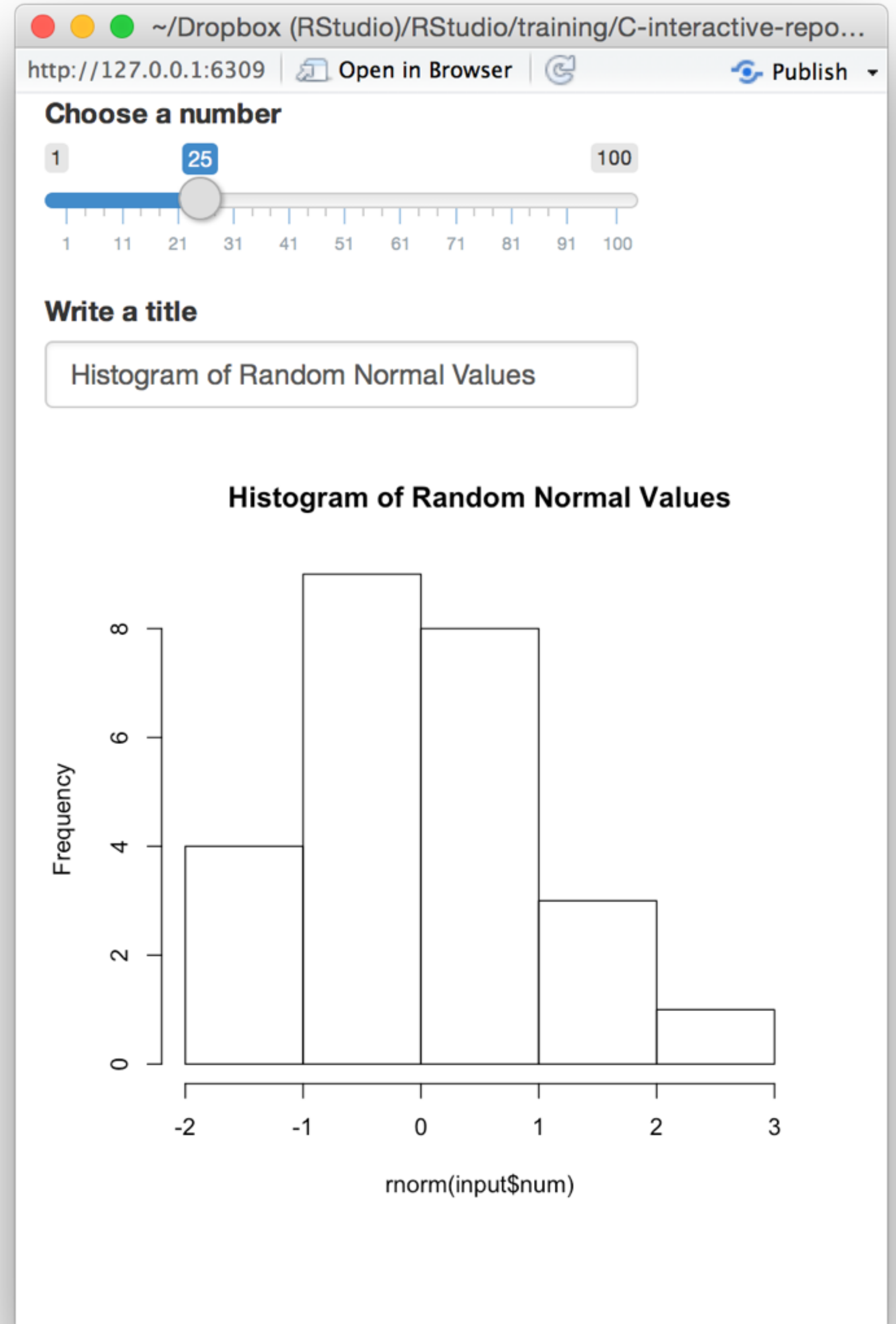
```
# 01-two-inputs
```

```
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)
```



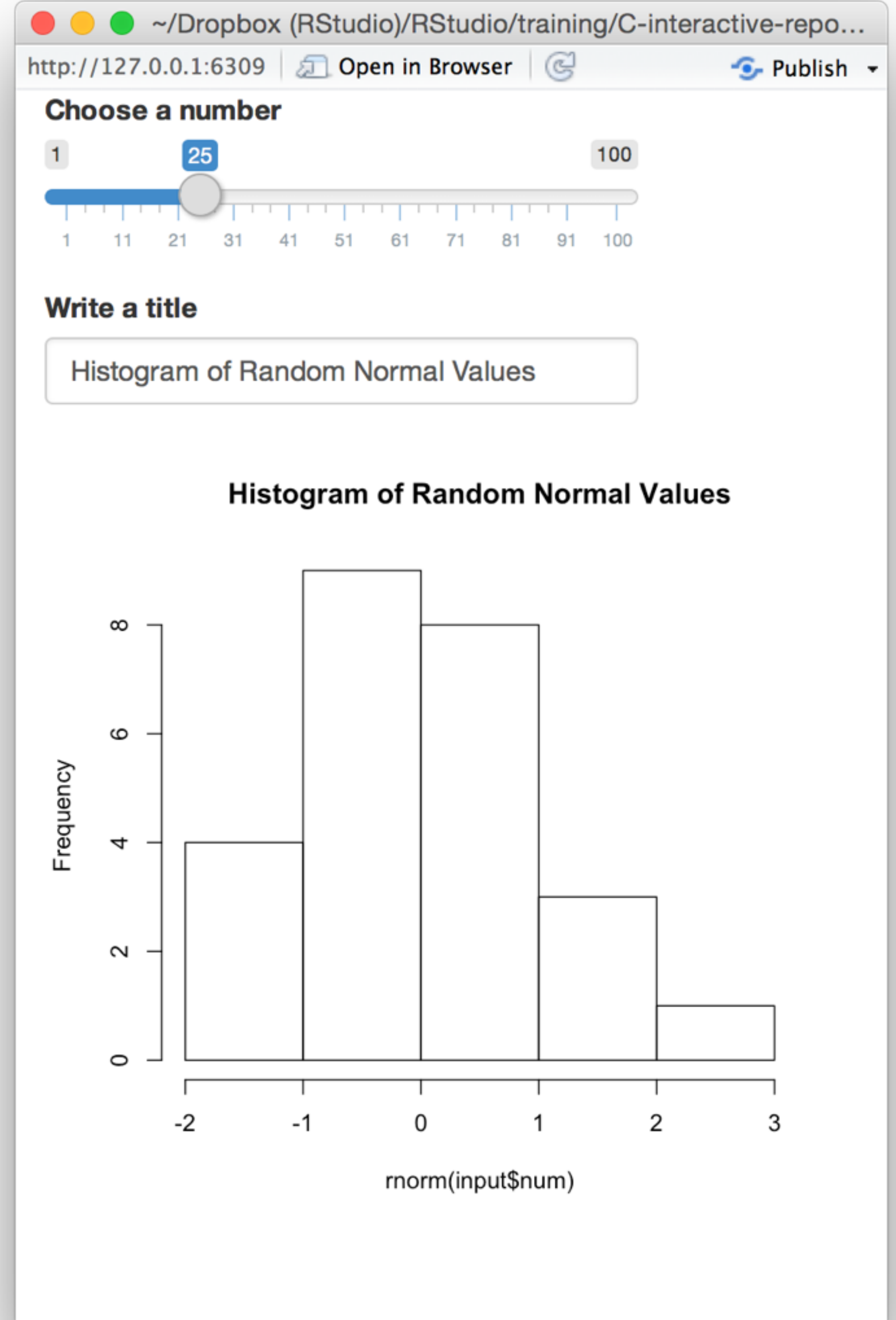
```
# 01-two-inputs
```

```
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)
```

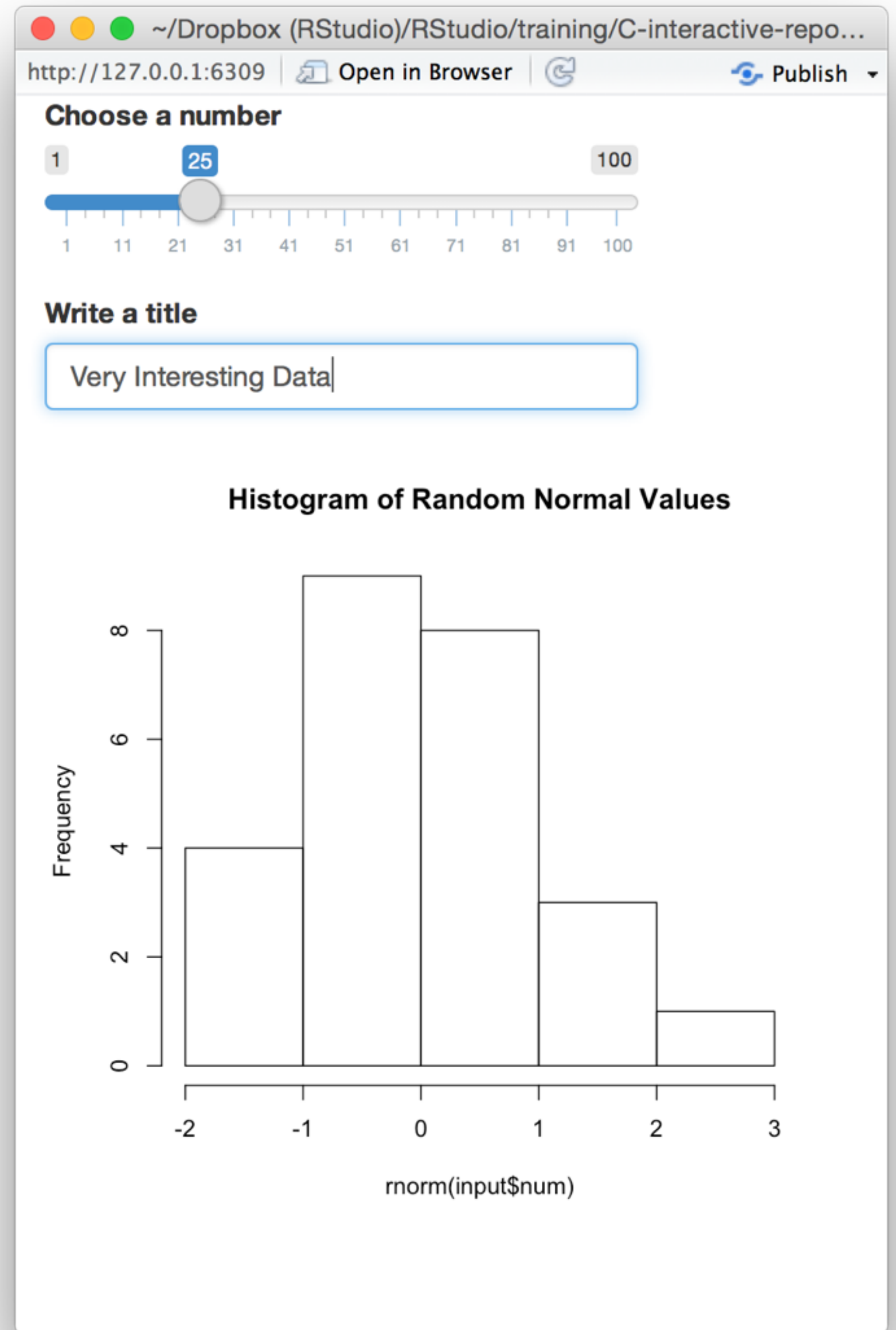




input\$num

input\$title

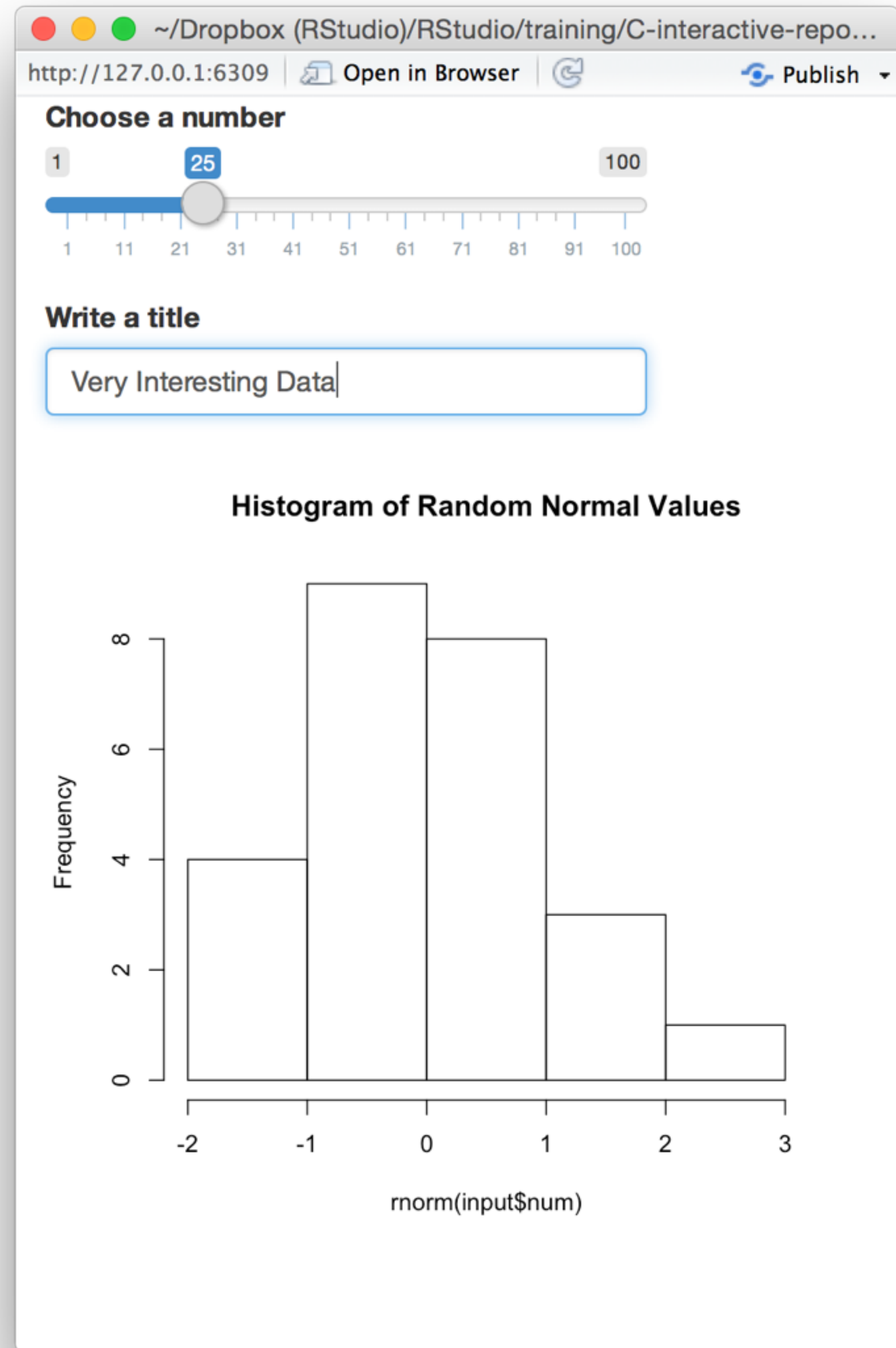
```
output$hist <- renderPlot({  
  hist(rnorm(input$num),  
       main = input$title)  
})
```



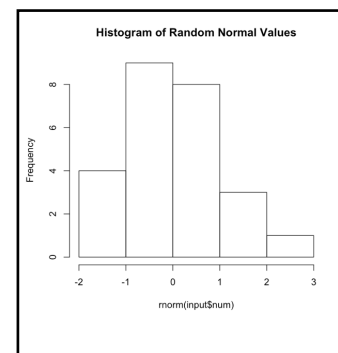
input\$num

input\$title

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



# Recap: `render*()`



**output\$**

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

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()**

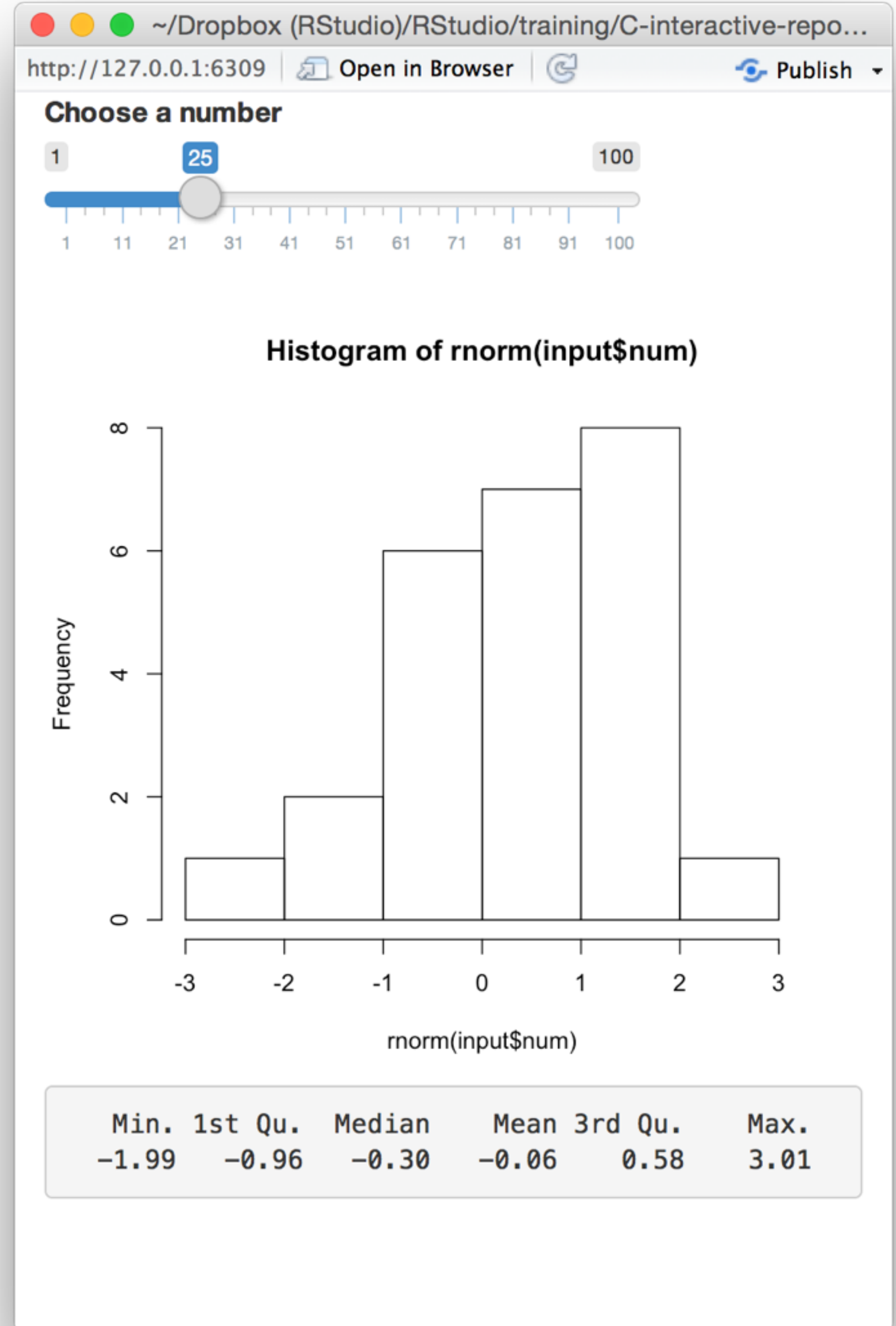
```
# 02-two-outputs
```

```
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)
```



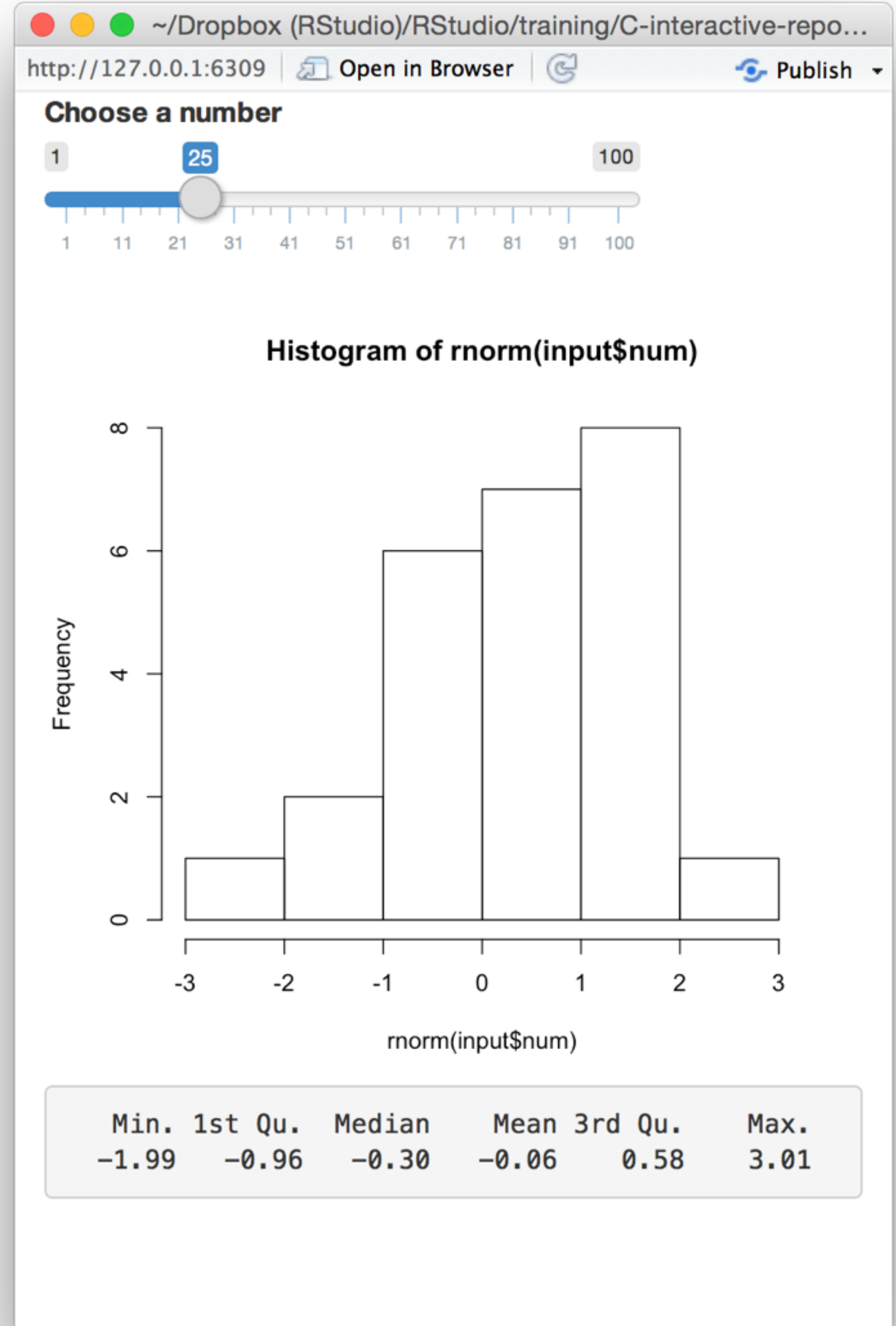
```
# 02-two-outputs
```

```
library(shiny)
```

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

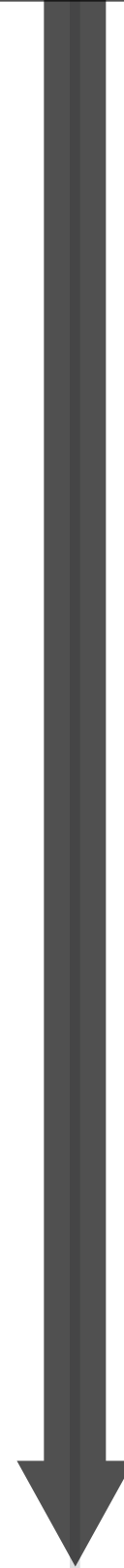
```
server <- function(input, output) {  
  output$hist <- renderPlot({  
    hist(rnorm(input$num))  
  })  
  output$stats <- renderPrint({  
    summary(rnorm(input$num))  
  })  
}
```

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





input\$num



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

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

~/Dropbox (RStudio)/RStudio/training/C-interactive-repo...  
http://127.0.0.1:6309 | Open in Browser | Publish

### Choose a number

1  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

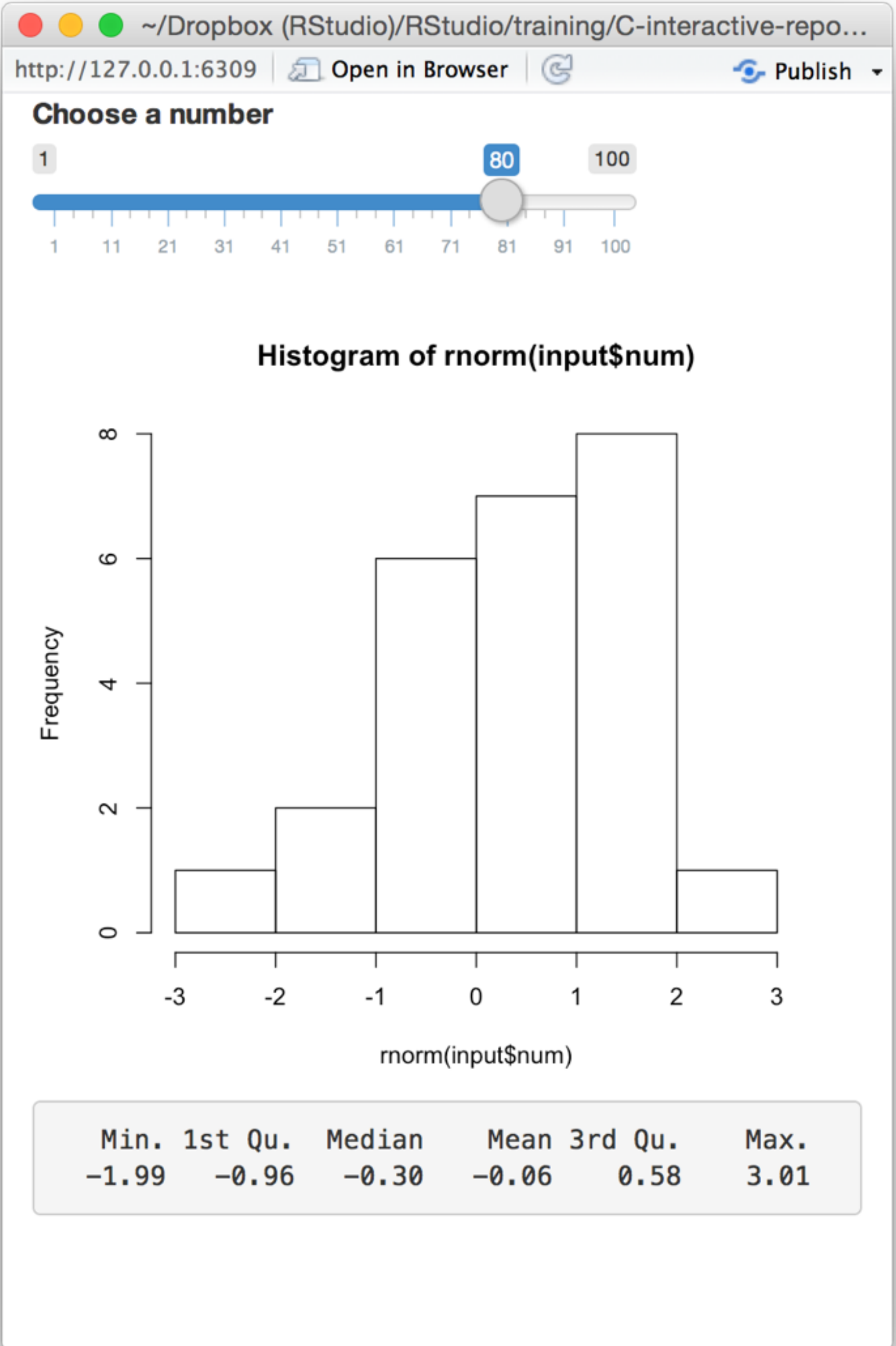
© CC 2015 RStudio, Inc.



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))  
  })
```

The screenshot shows an R Shiny application interface. At the top, there is a slider titled "Choose a number" with a range from 1 to 100. The current value is 80. Below the slider is a histogram titled "Histogram of rnorm(input\$num)". The x-axis is labeled "rnorm(input\$num)" and ranges from -2 to 3. The y-axis is labeled "Frequency" and ranges from 0 to 15. Below the histogram is a summary table for the data.

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

1 11 21 31 41 51 61 71 81 91 100

Histogram of rnorm(input\$num)

Frequency

rnorm(input\$num)

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

© CC 2015 RStudio, Inc.

# 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) {
```

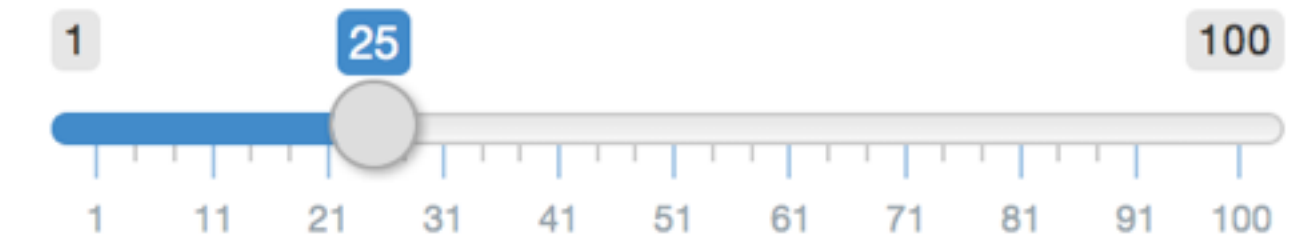
```
  output$hist <- renderPlot({  
    hist(rnorm(input$num))  
  })
```

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

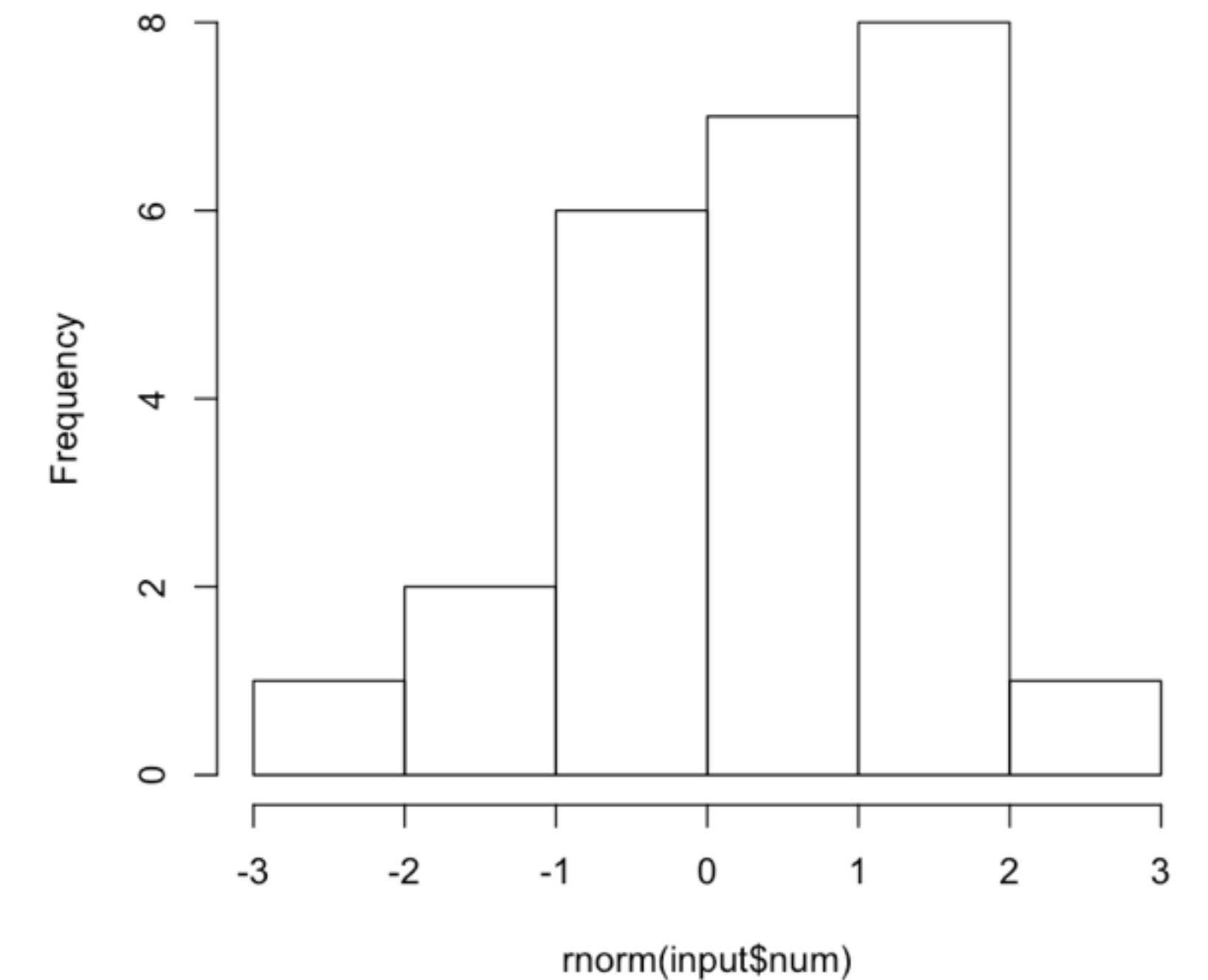
```
}
```

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

Choose a number



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



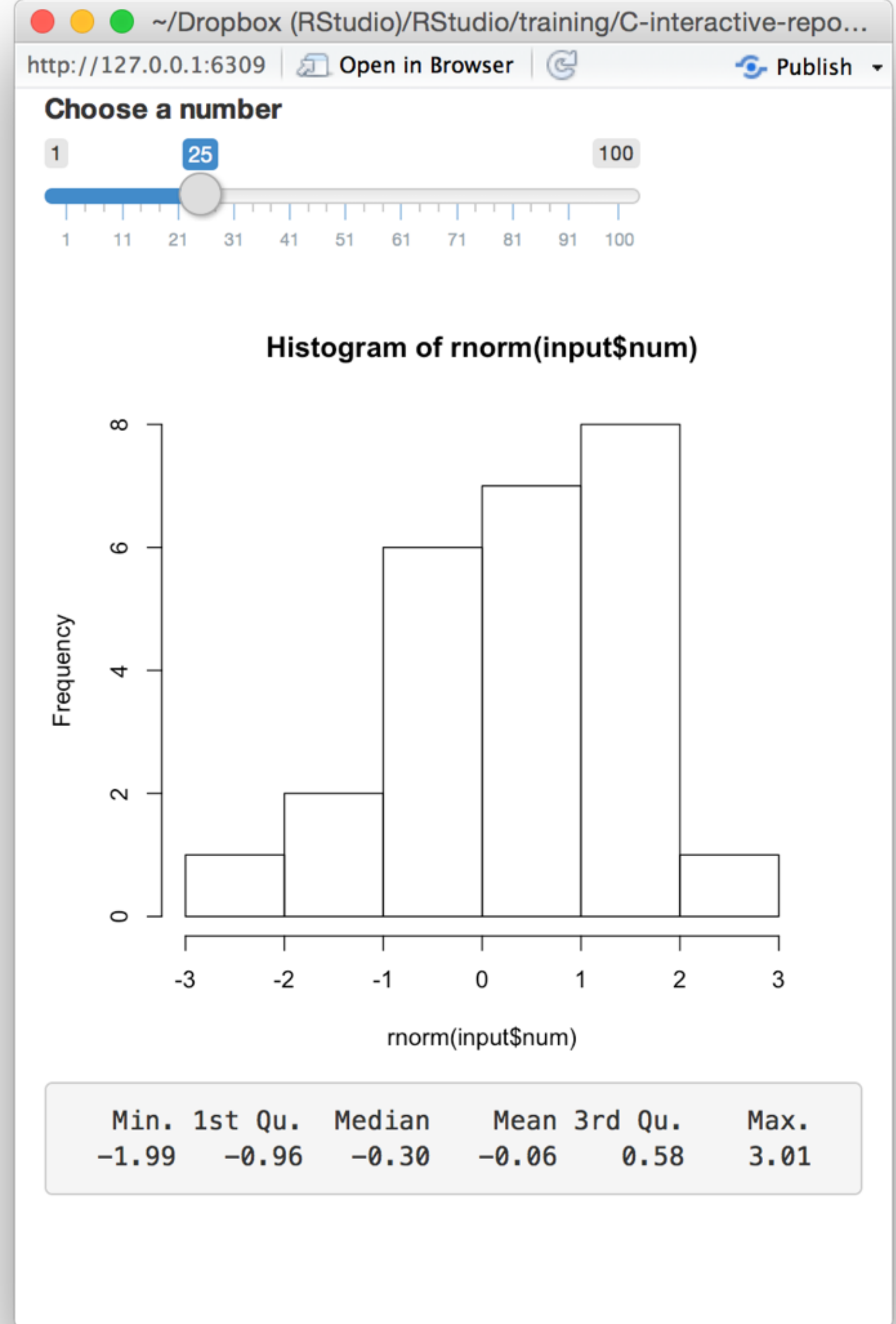
```
# 02-two-outputs
```

```
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

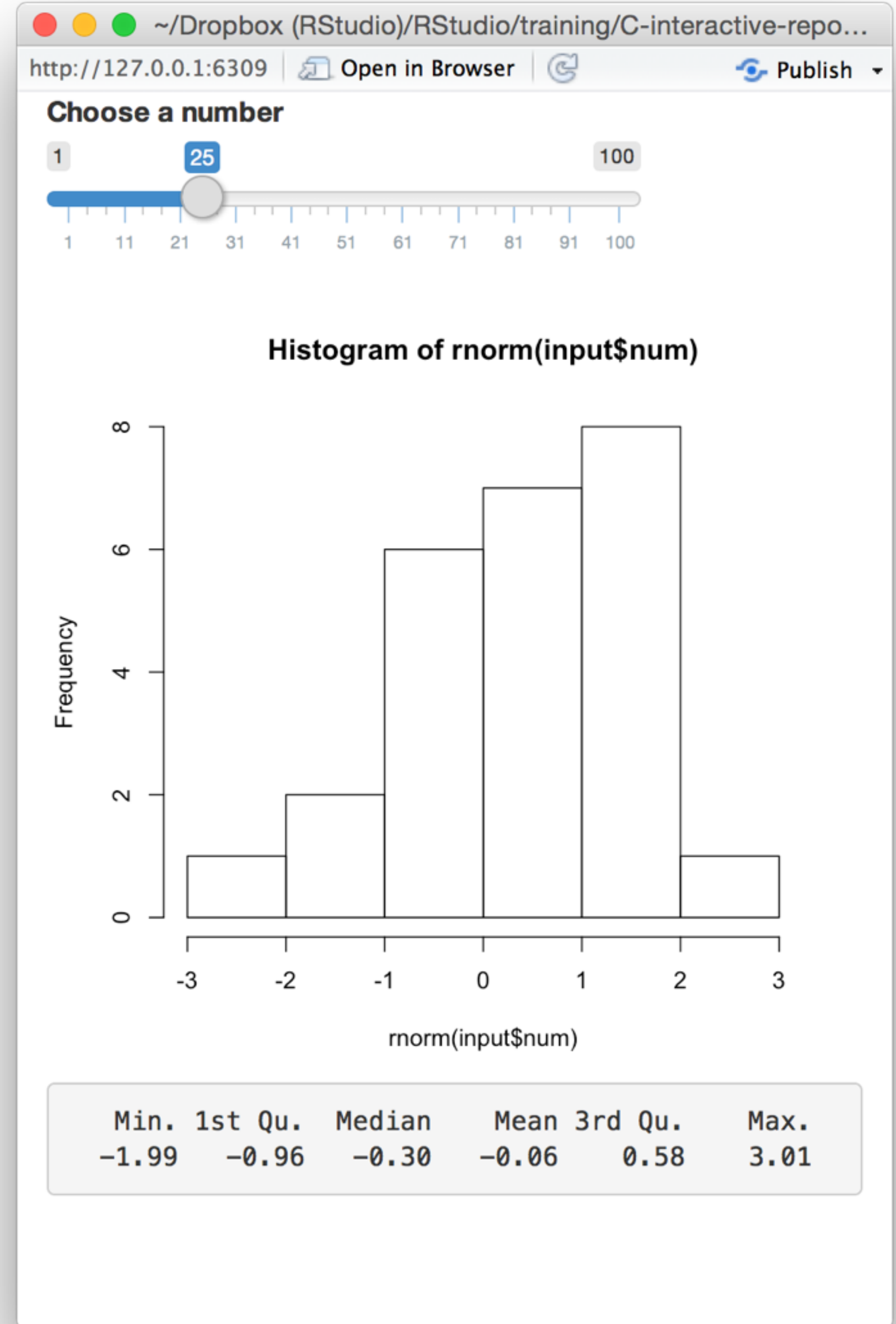
library(shiny)

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)

```

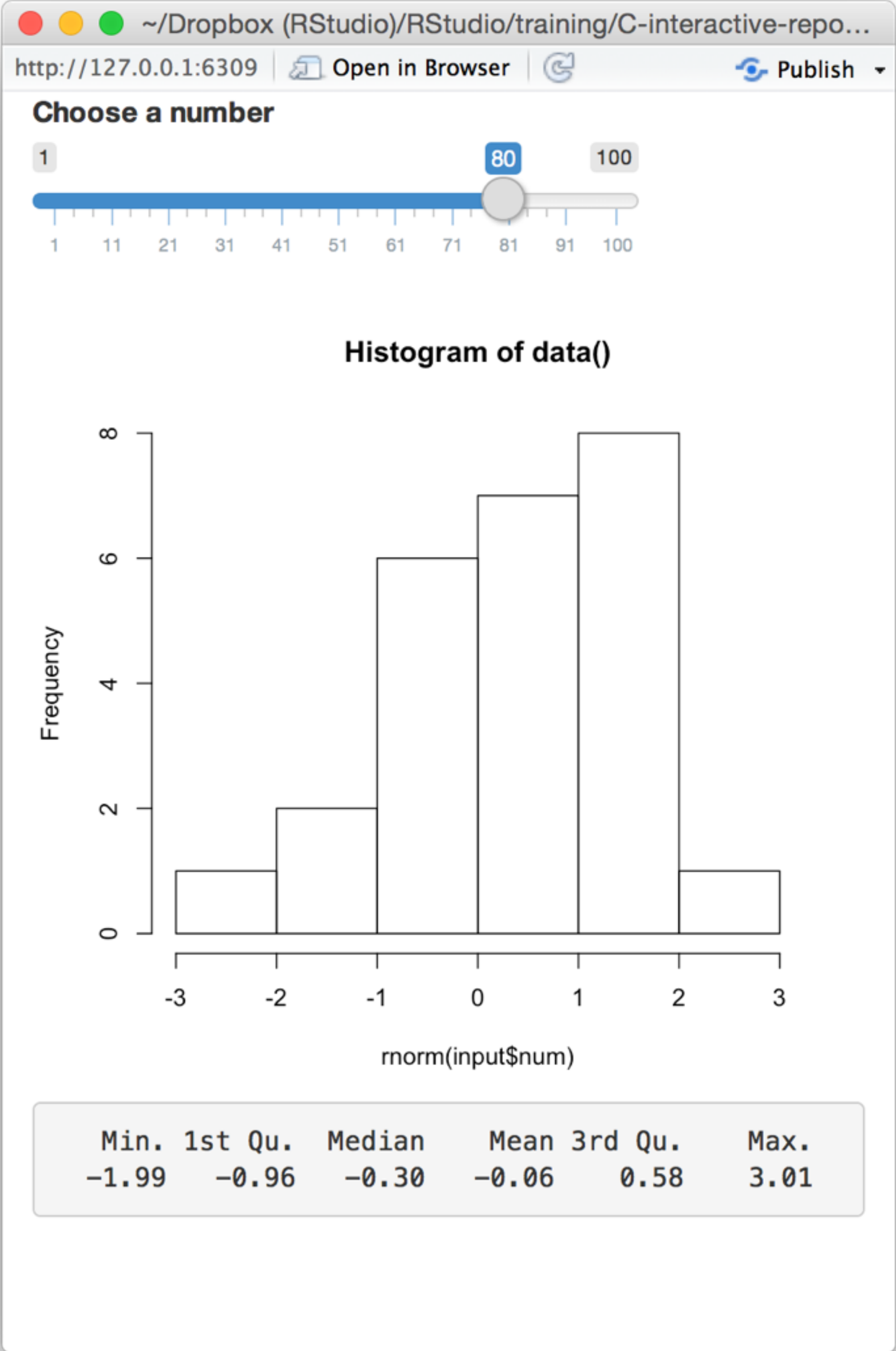


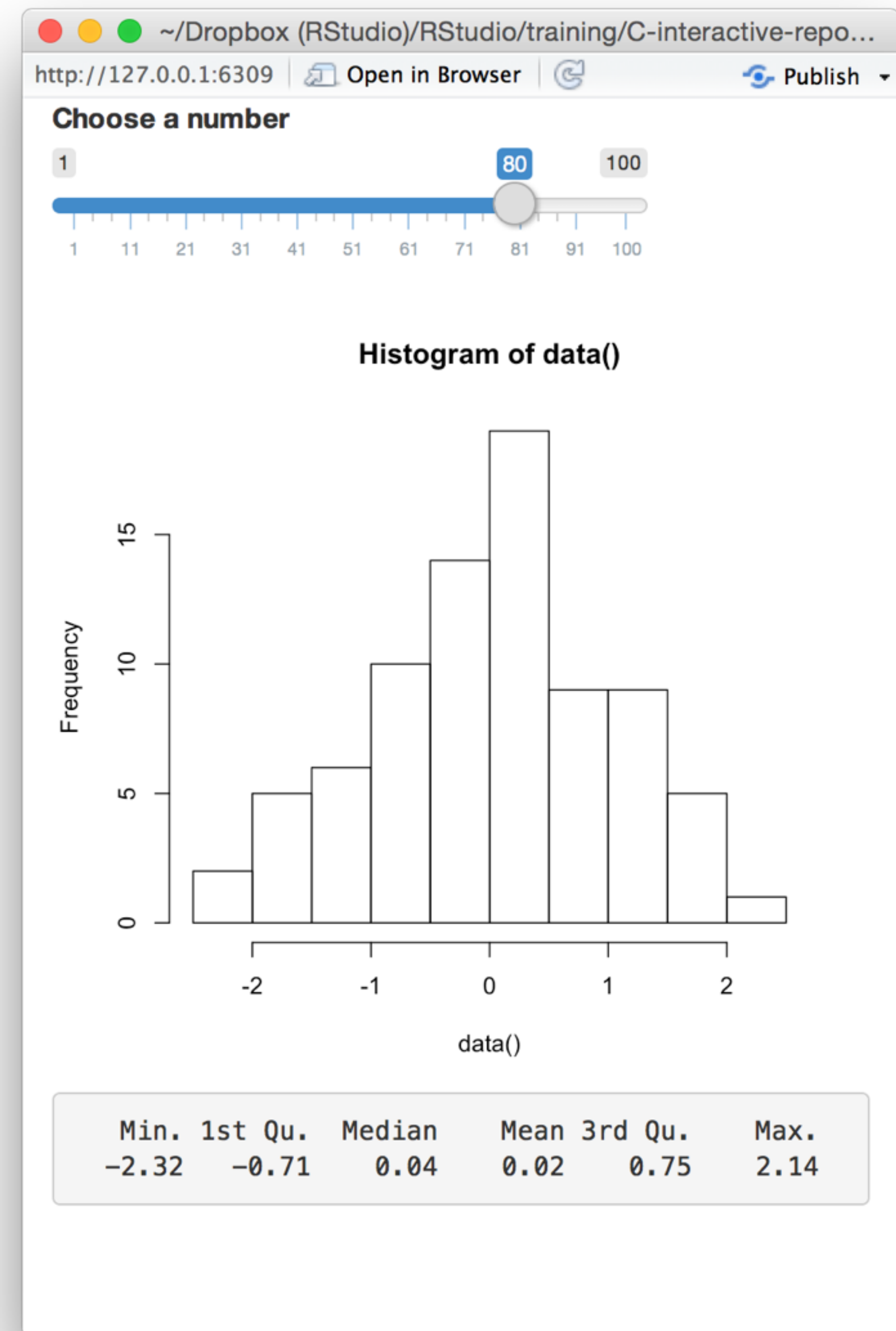
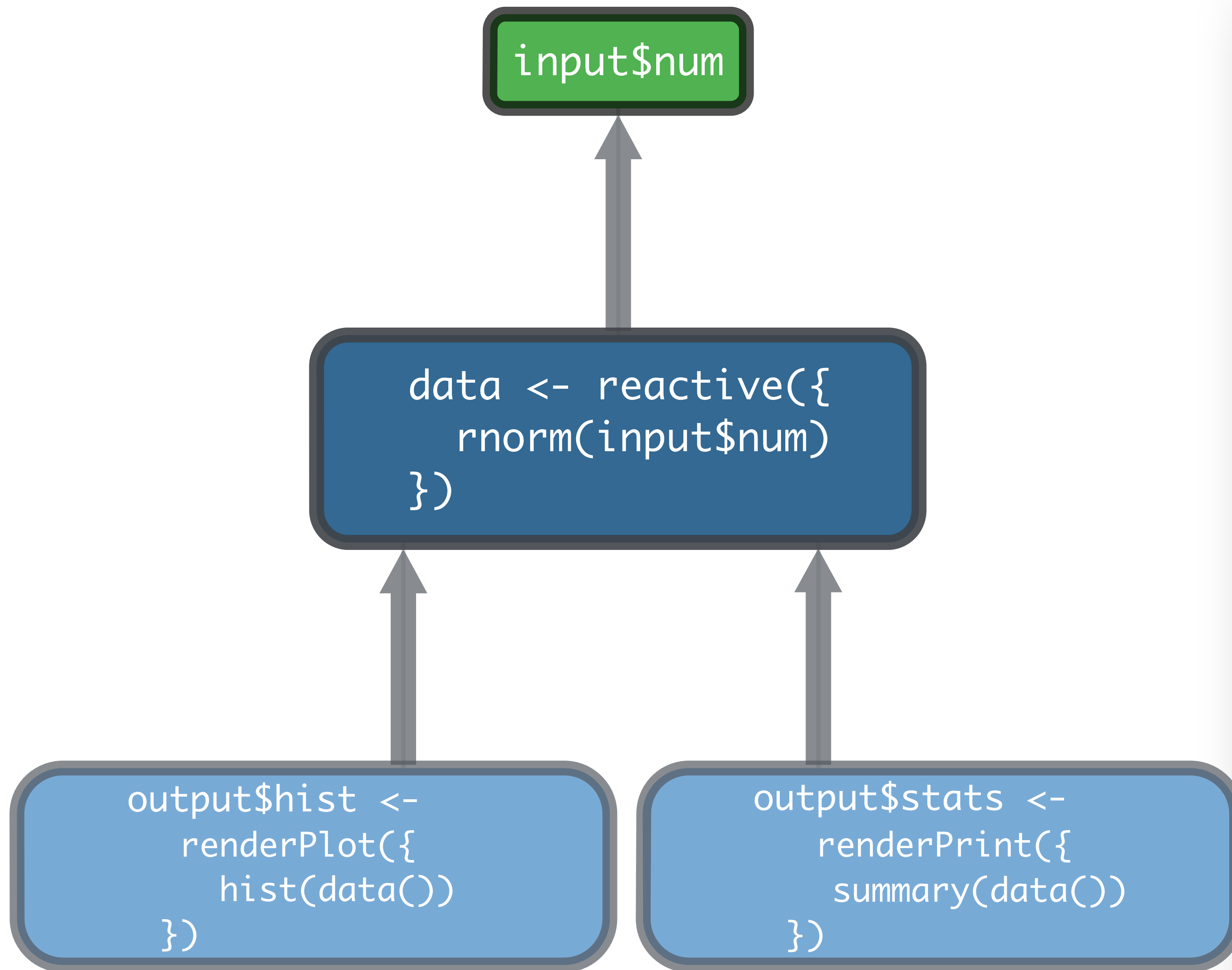
input\$num

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

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

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





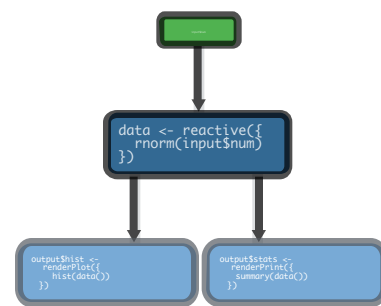
# A reactive expression is special in two ways

```
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),
  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)

```

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



~/Dropbox (RStudio)/RStudio/training/C-interactive-repo...  
 http://127.0.0.1:6309 | Open in Browser | Publish

**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**

Bin Range	Frequency
-2 to -1	4
-1 to 0	9
0 to 1	8
1 to 2	3
2 to 3	1



# 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

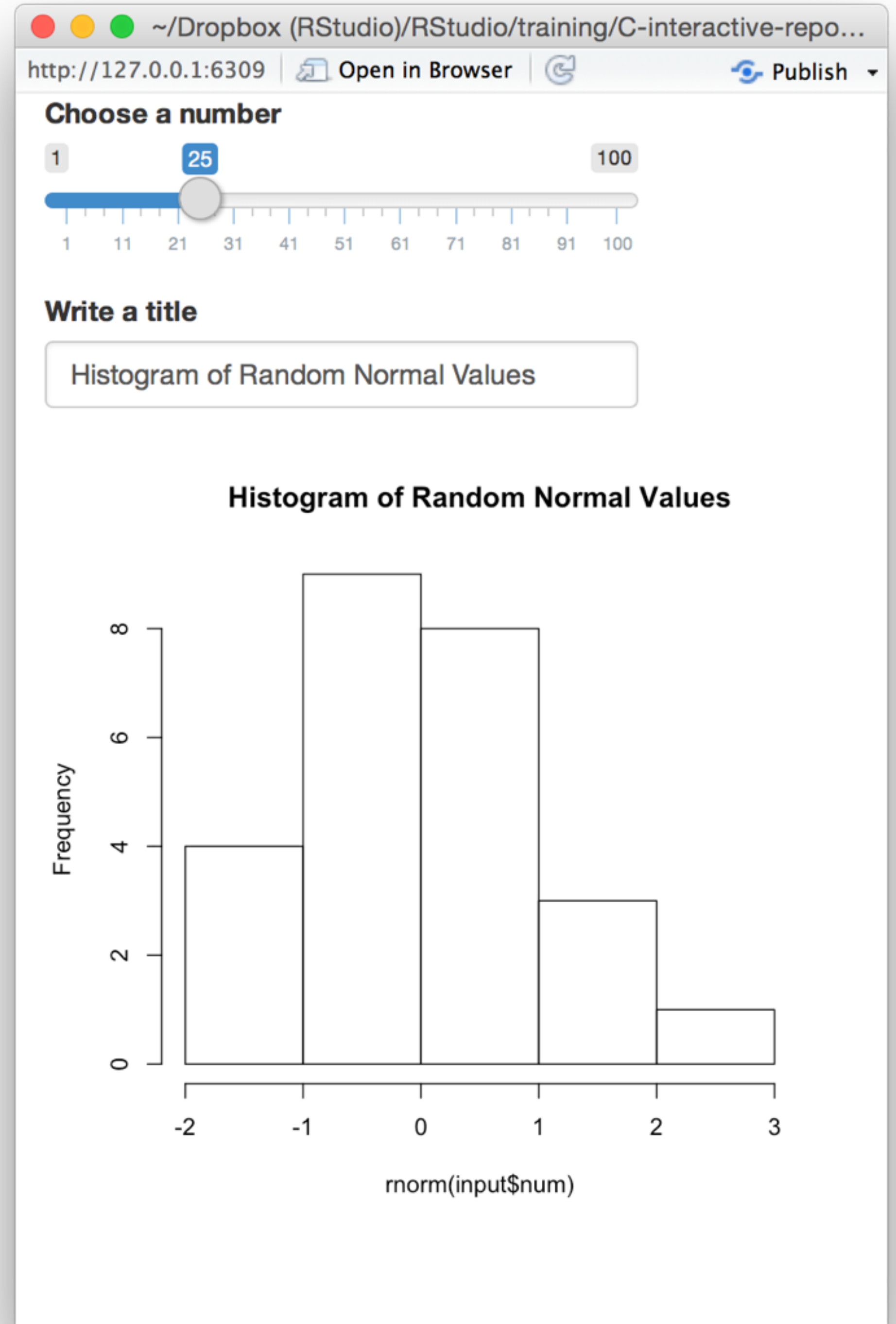
library(shiny)

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

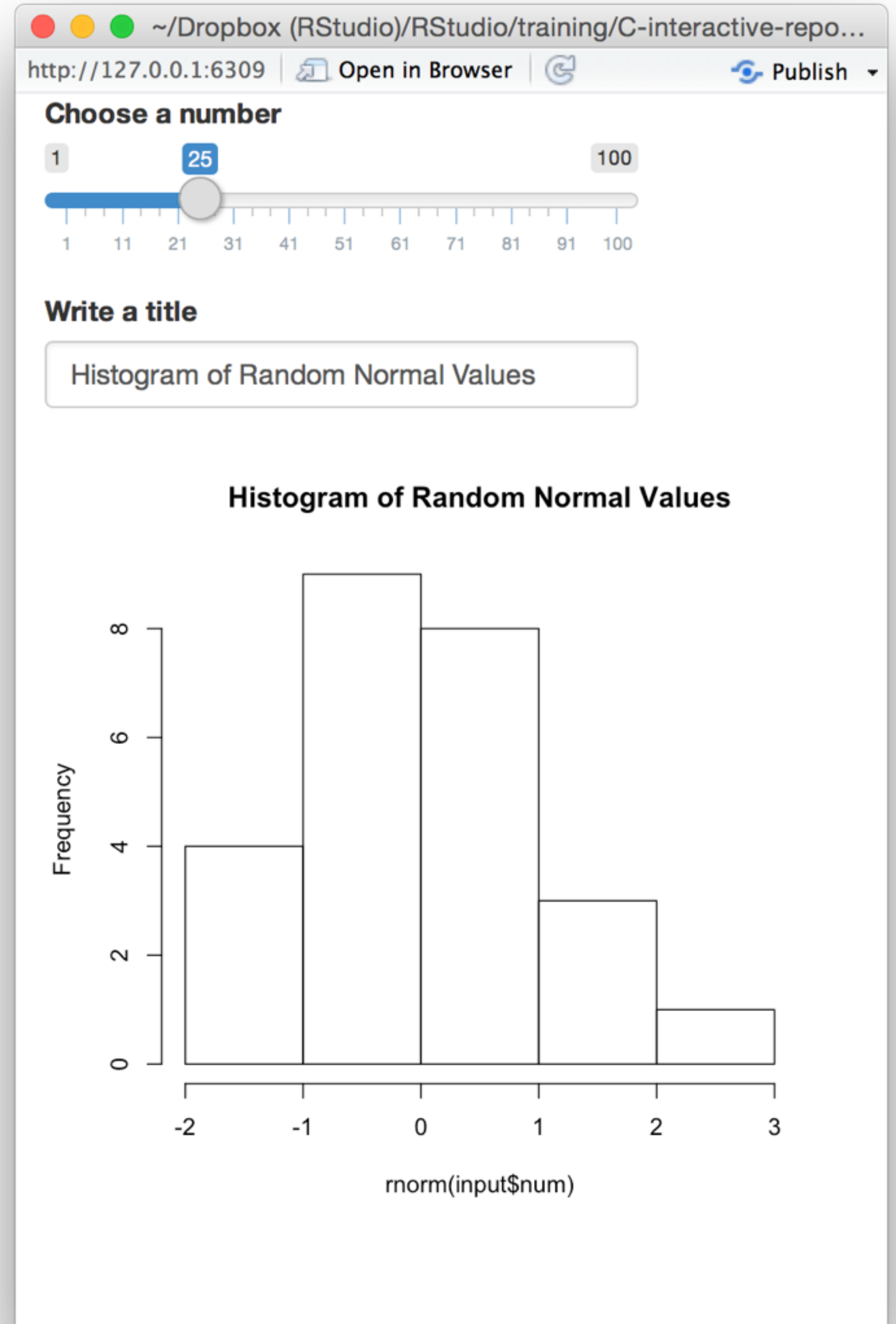
library(shiny)

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)

```

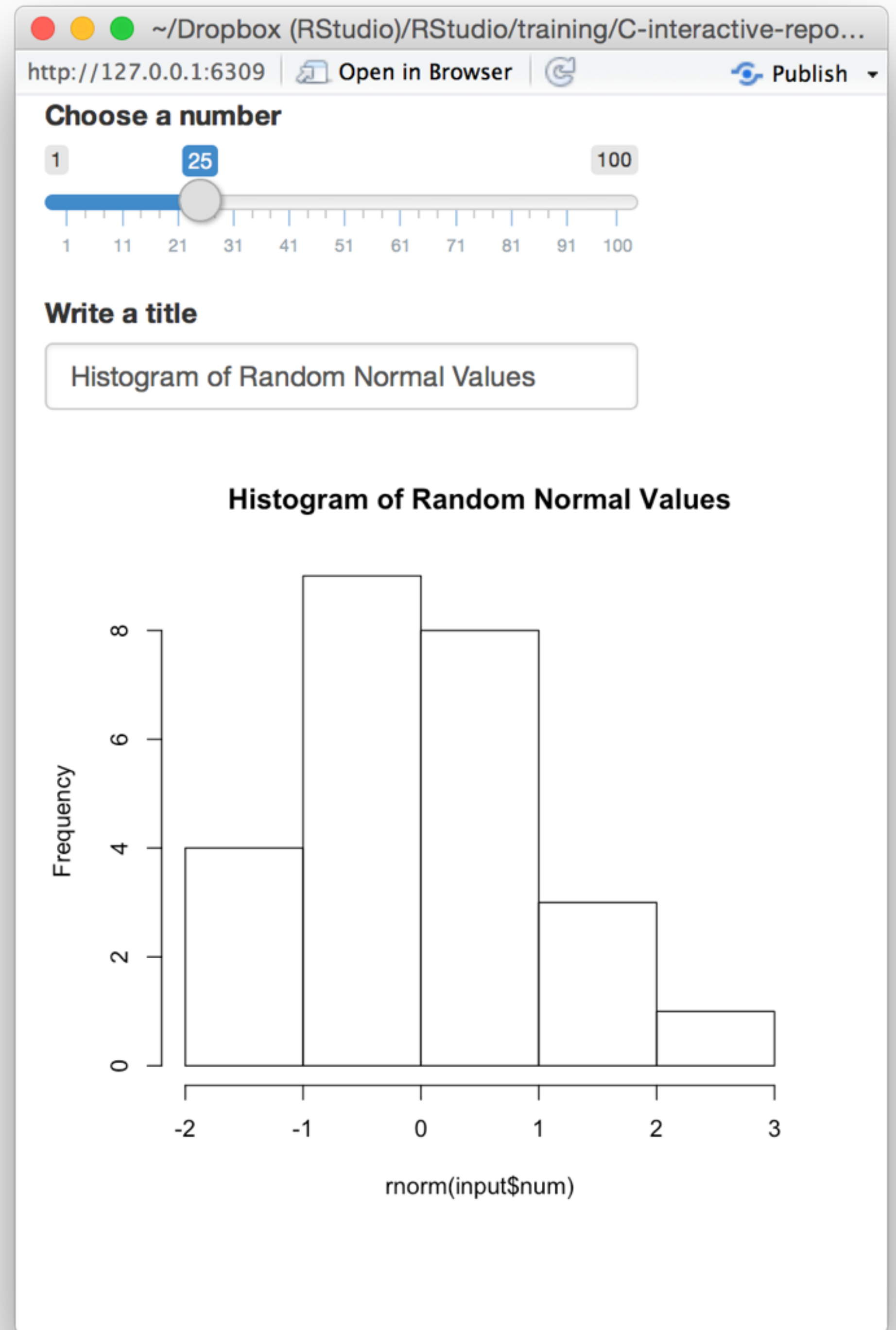


input\$num

input\$title



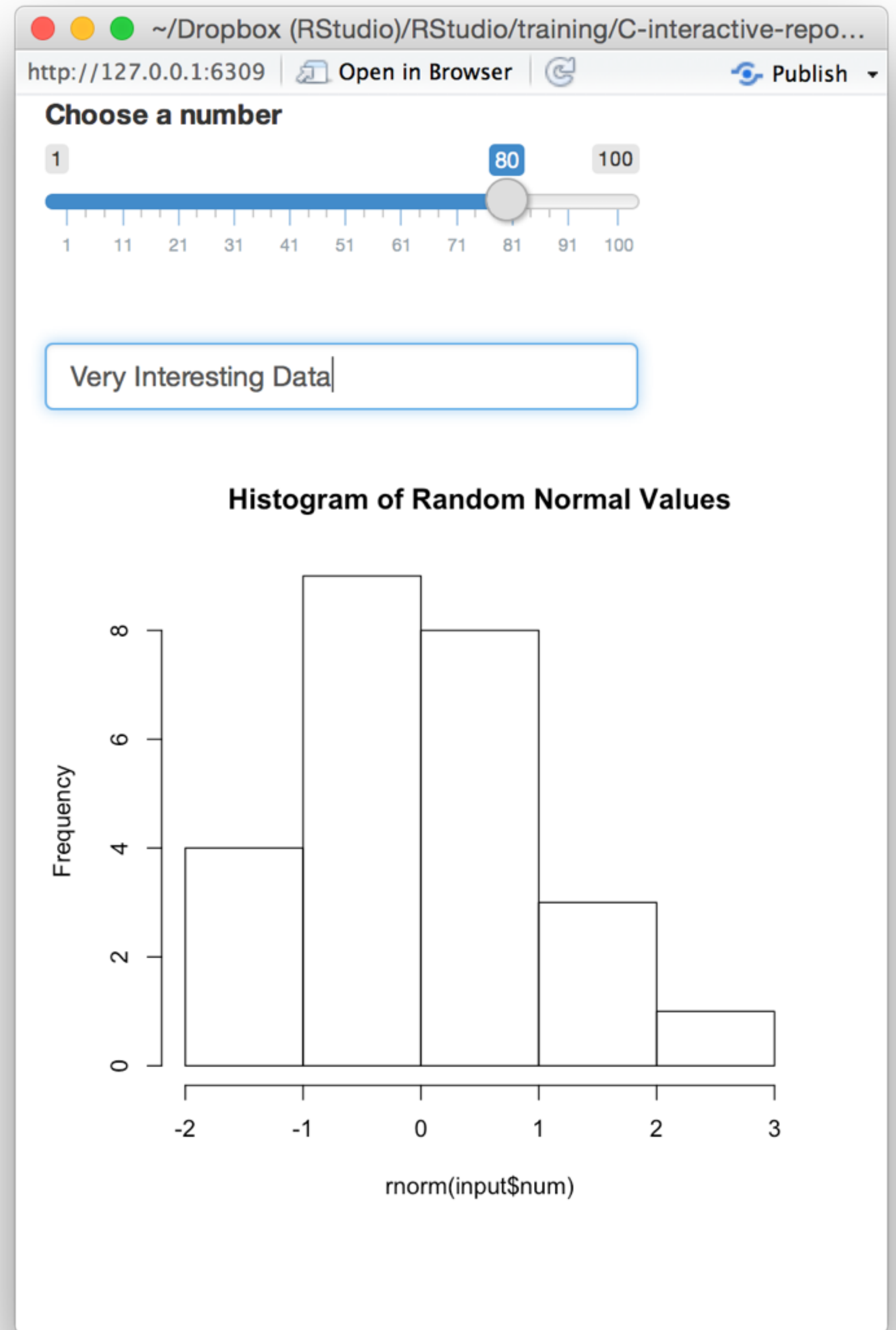
```
output$hist <- renderPlot({  
  hist(rnorm(input$num),  
       main = isolate(input$title))  
})
```

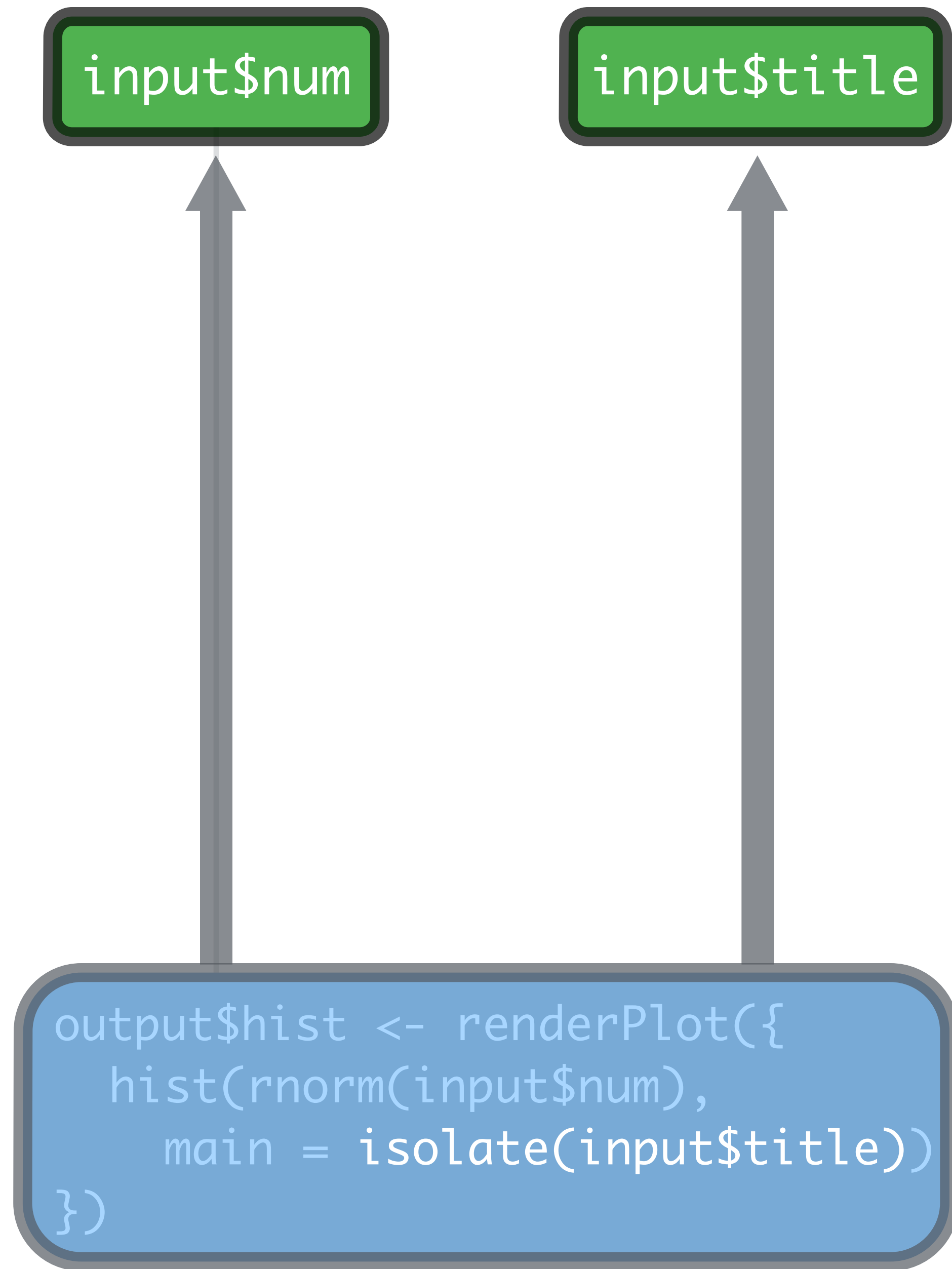


input\$num

input\$title

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





~/Dropbox (RStudio)/RStudio/training/C-interactive-repo...  
http://127.0.0.1:6309 | Open in Browser | Publish

### Choose a number

1 80 100

1 11 21 31 41 51 61 71 81 91 100

Very Interesting Data

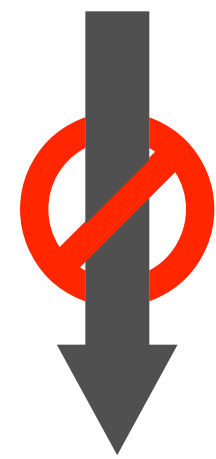
### Histogram of Random Normal Values

Frequency

rnorm(input\$num)

Bin Range	Frequency
-2.0 to -1.0	4
-1.0 to 0.0	9
0.0 to 1.0	8
1.0 to 2.0	3
2.0 to 3.0	1

# Recap: isolate()



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

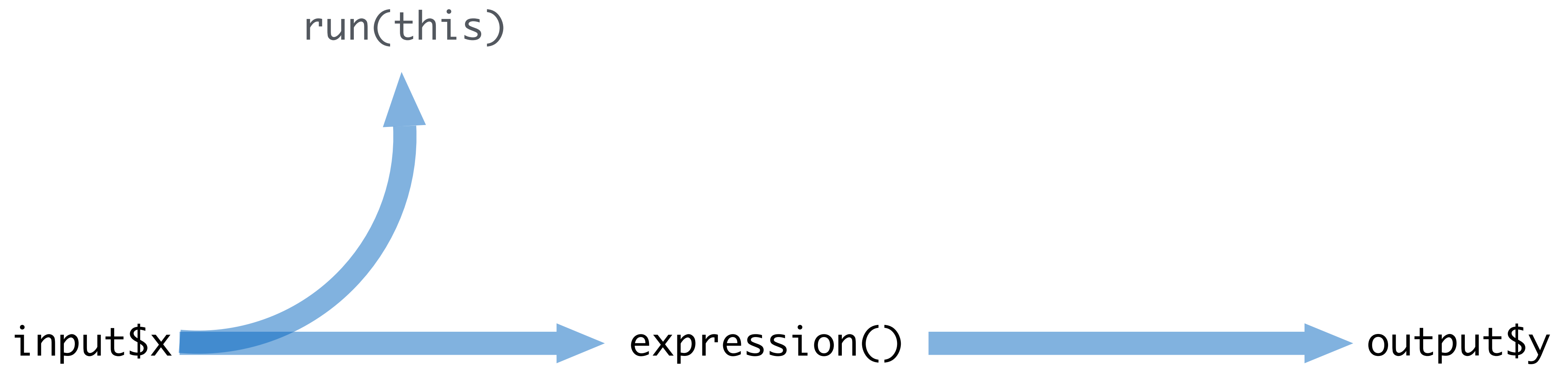


Use isolate() to treat reactive values like normal R values



**Trigger code**  
**with `observeEvent()`**





# 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

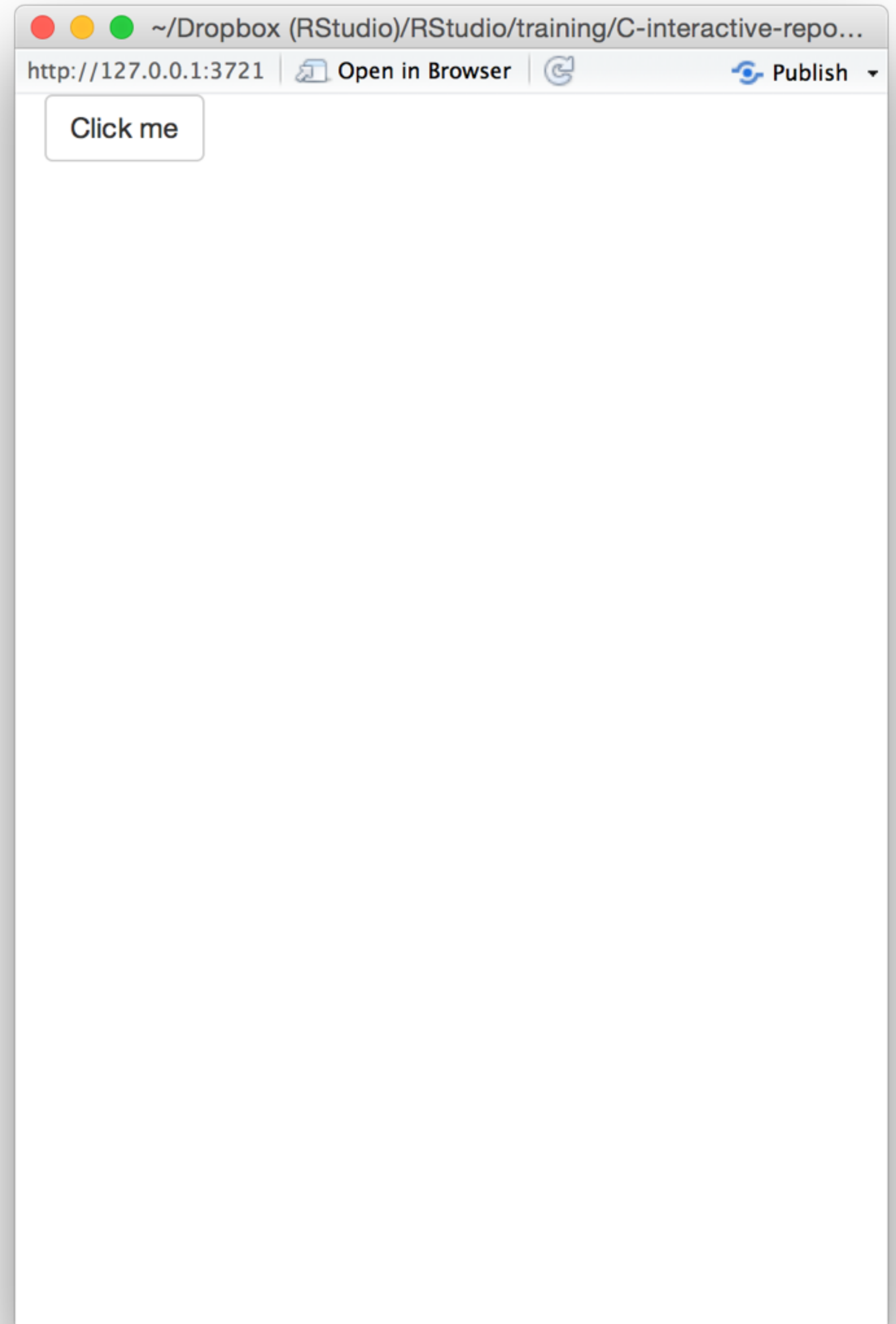
```
# 05-actionButton
```

```
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()

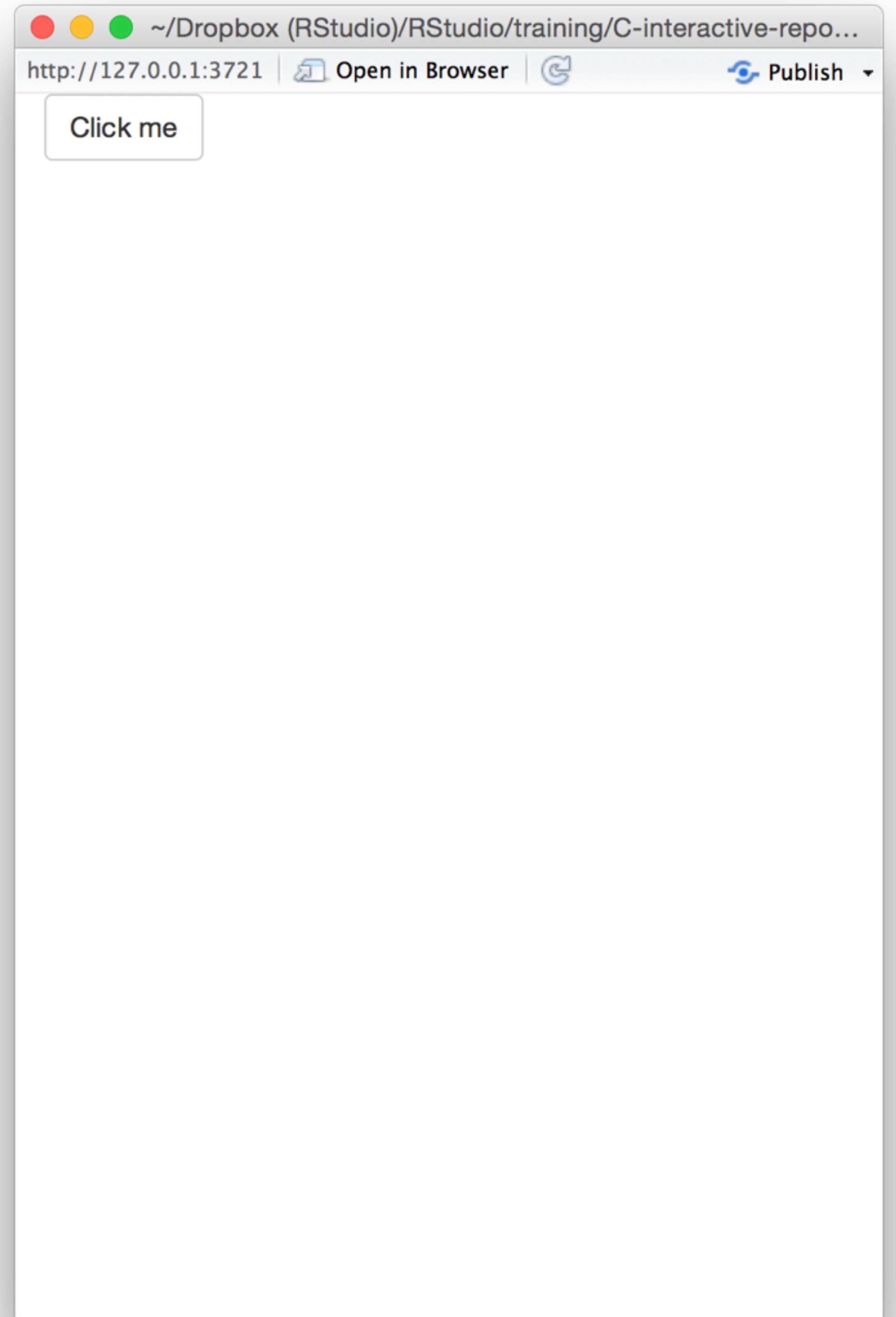
```
# 05-actionButton
```

```
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 displaying the Shiny website. The browser's address bar shows the URL `shiny.rstudio.com/articles/action-buttons.html`. The page header features the Shiny logo and a search bar. A left-hand navigation menu includes links for OVERVIEW, TUTORIAL, ARTICLES (which is highlighted), GALLERY, REFERENCE, DEPLOY, and HELP. The main content area is titled "Using Action Buttons" and includes a sub-header "How action buttons work". The text explains that action buttons and links are used with `observeEvent()` or `eventReactive()`. It lists two arguments for the `actionButton()` and `actionLink()` functions: `inputId` and `label`. A code block shows the following R code: 

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

 Below the code, there is a visual example of an action button labeled "An action button".

Shiny by RStudio

Shiny - Using Action Buttons

shiny.rstudio.com/articles/action-buttons.html

Garrett

Search

OVERVIEW

TUTORIAL

ARTICLES

GALLERY

REFERENCE

DEPLOY

HELP

## Using Action Buttons

ADDED: 26 MAR 2015

This article describes five patterns to use with Shiny's [action buttons](#) and [action links](#). Action buttons and action links are different from other Shiny widgets because they are intended to be used exclusively with `observeEvent()` or `eventReactive()`.

### How action buttons work

Create an action button with `actionButton()` and an action link with `actionLink()`. Each of these functions takes two arguments:

- `inputId` - the ID of the button or link
- `label` - the label to display in the button or link

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

An action button appears as a button in your app.

An action link appears as a hyperlink, but behaves in the same way as an action button.

Inc.

# observe()

Also triggers code to run on server.

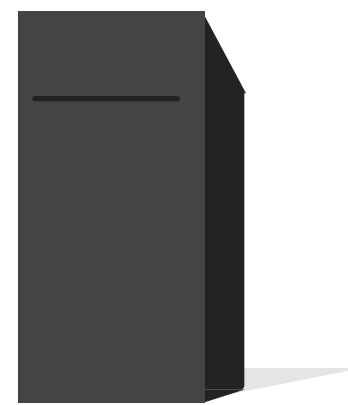
Uses same syntax as `render*()`, `reactive()`, and `isolate()`

```
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

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),
```

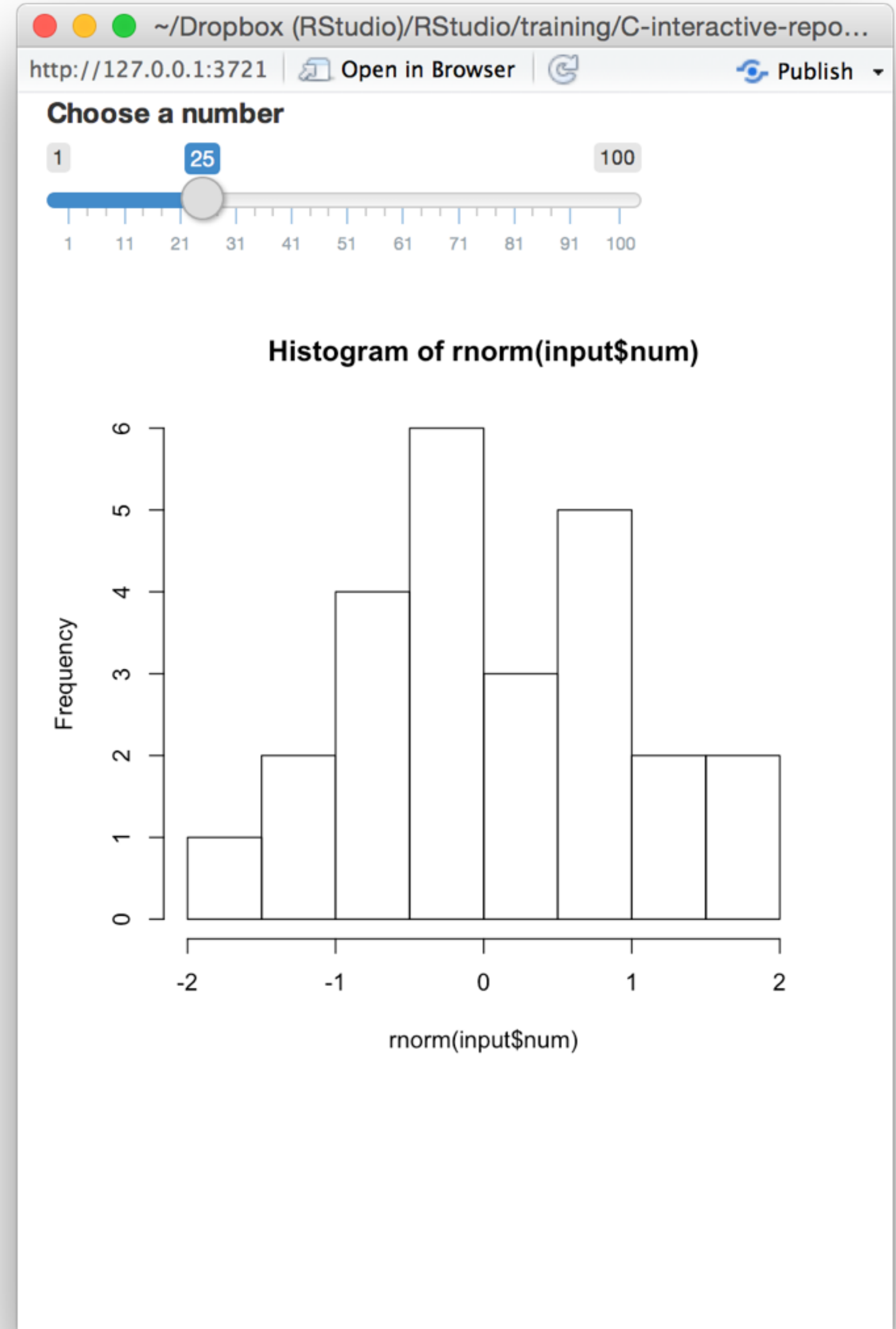
```
  plotOutput("hist")  
)
```

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

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

```
}
```

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



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

```
# 07-eventReactive

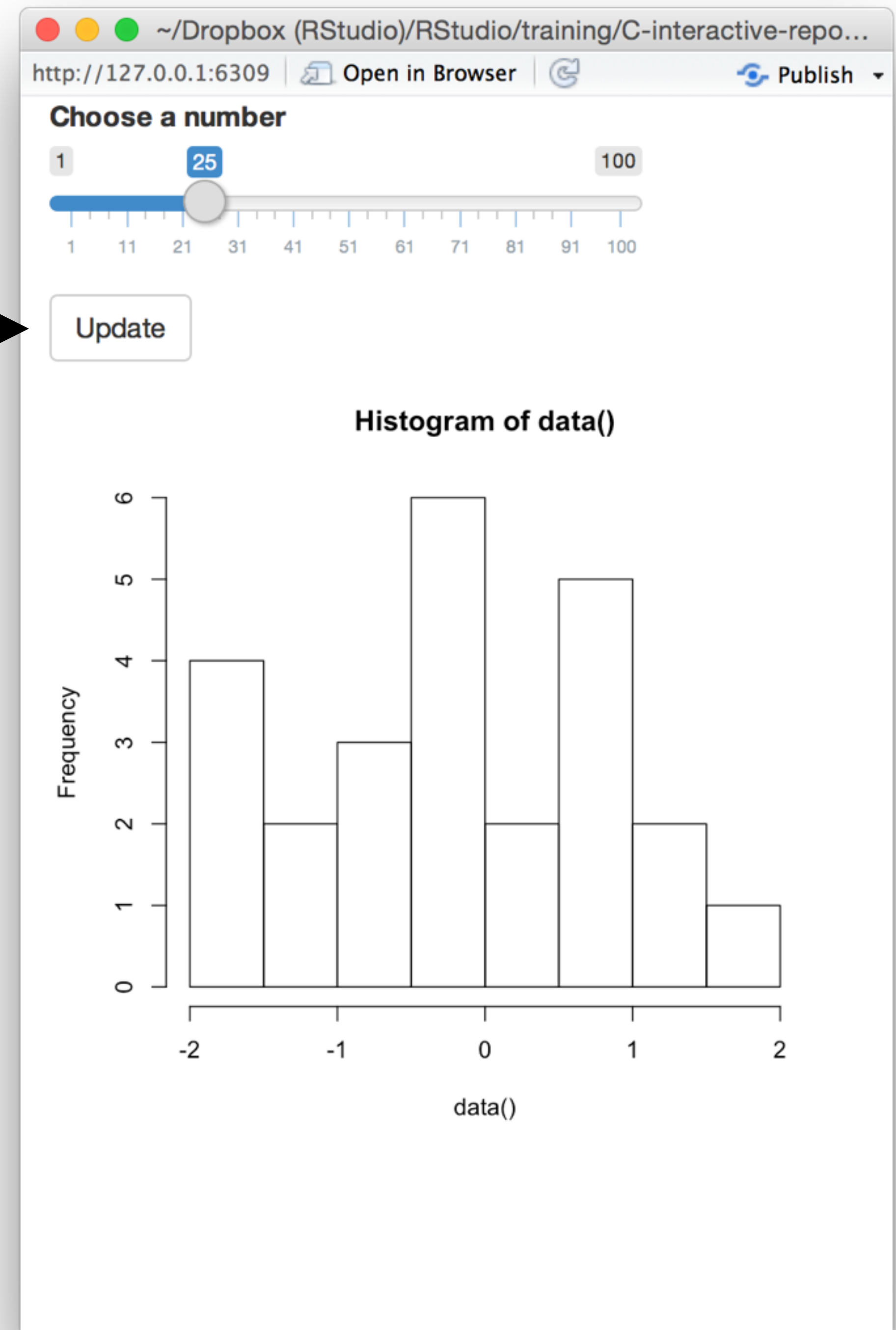
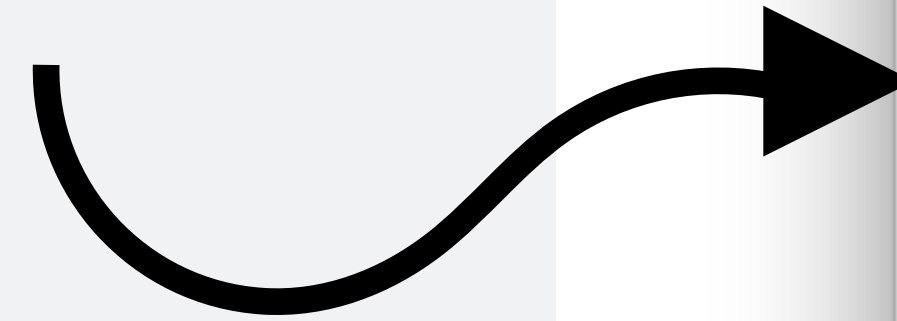
library(shiny)

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)
```





# eventReactive()

A reactive expression that only responds to specific values

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

reactive value(s) to  
respond to

(expression invalidates ONLY  
when this value changes)

code used to build (and  
rebuild) object

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



```
# 07-eventReactive
```

```
library(shiny)
```

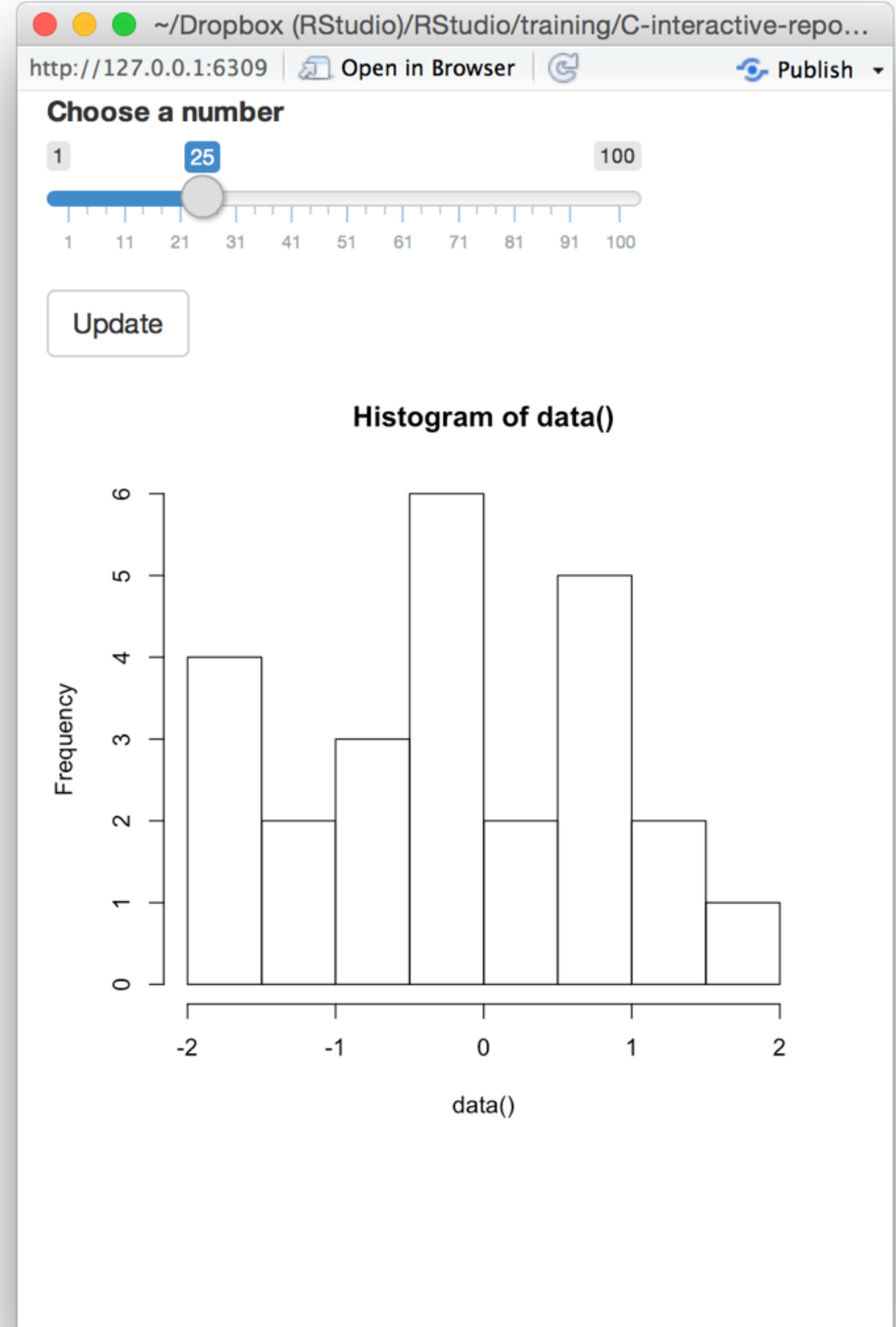
```
ui <- fluidPage(  
  sliderInput(inputId = "num",  
    label = "Choose a number",  
    value = 25, min = 1, max = 100),  
  actionButton(inputId = "go",  
    label = "Update"),  
  plotOutput("hist")  
)
```

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

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

```
}
```

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



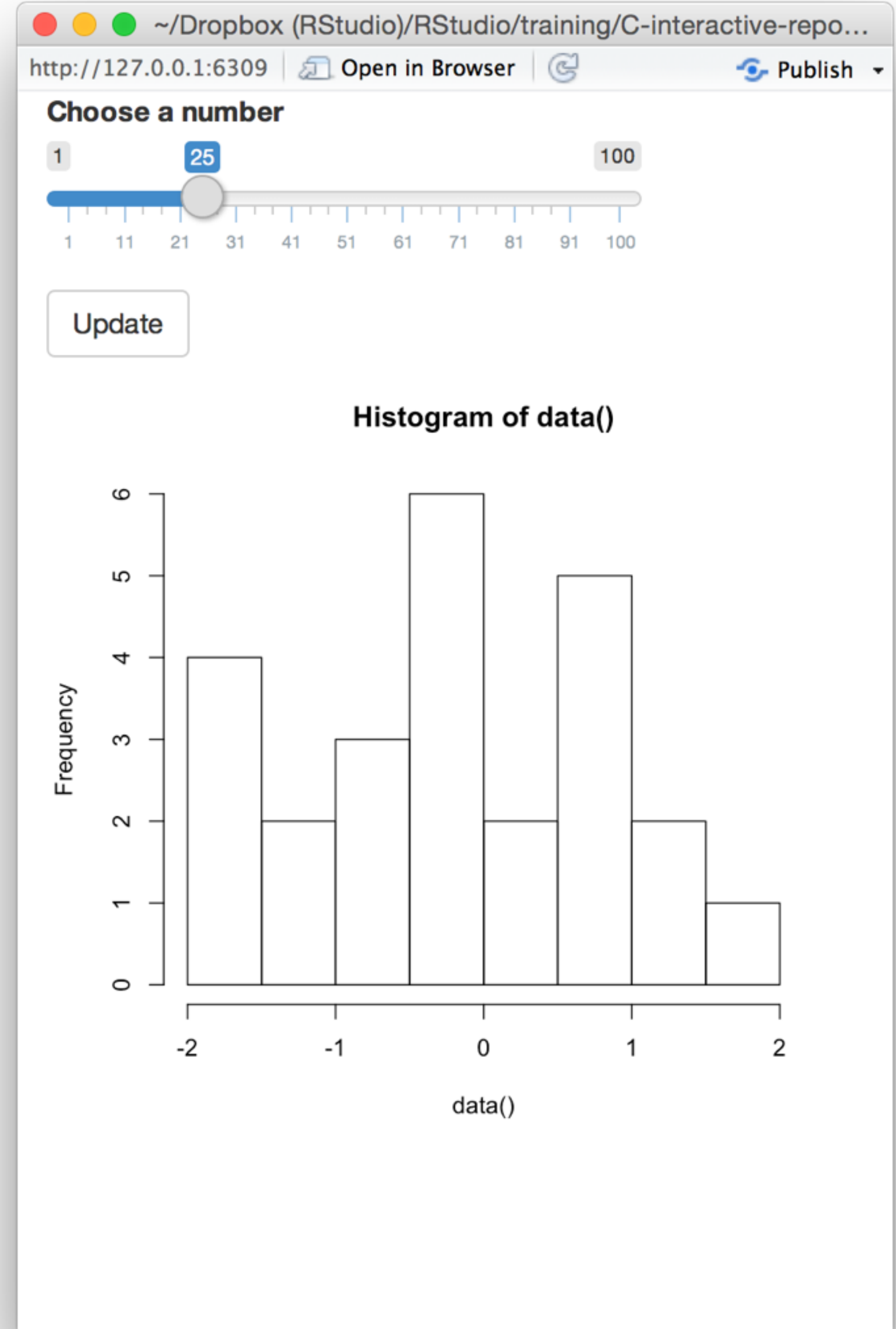
```
# 07-eventReactive
```

```
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)
```



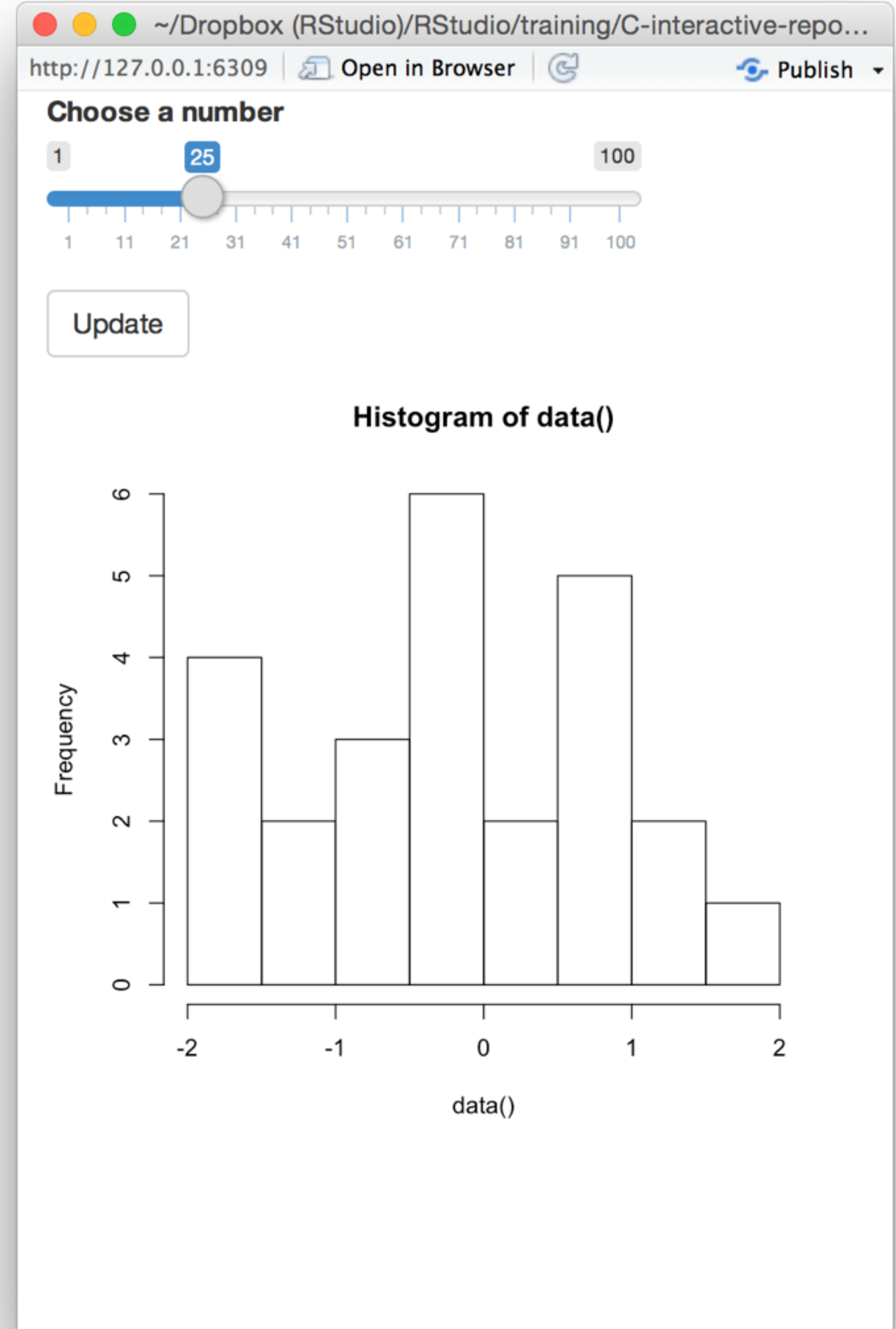
```
# 07-eventReactive
```

```
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)
```



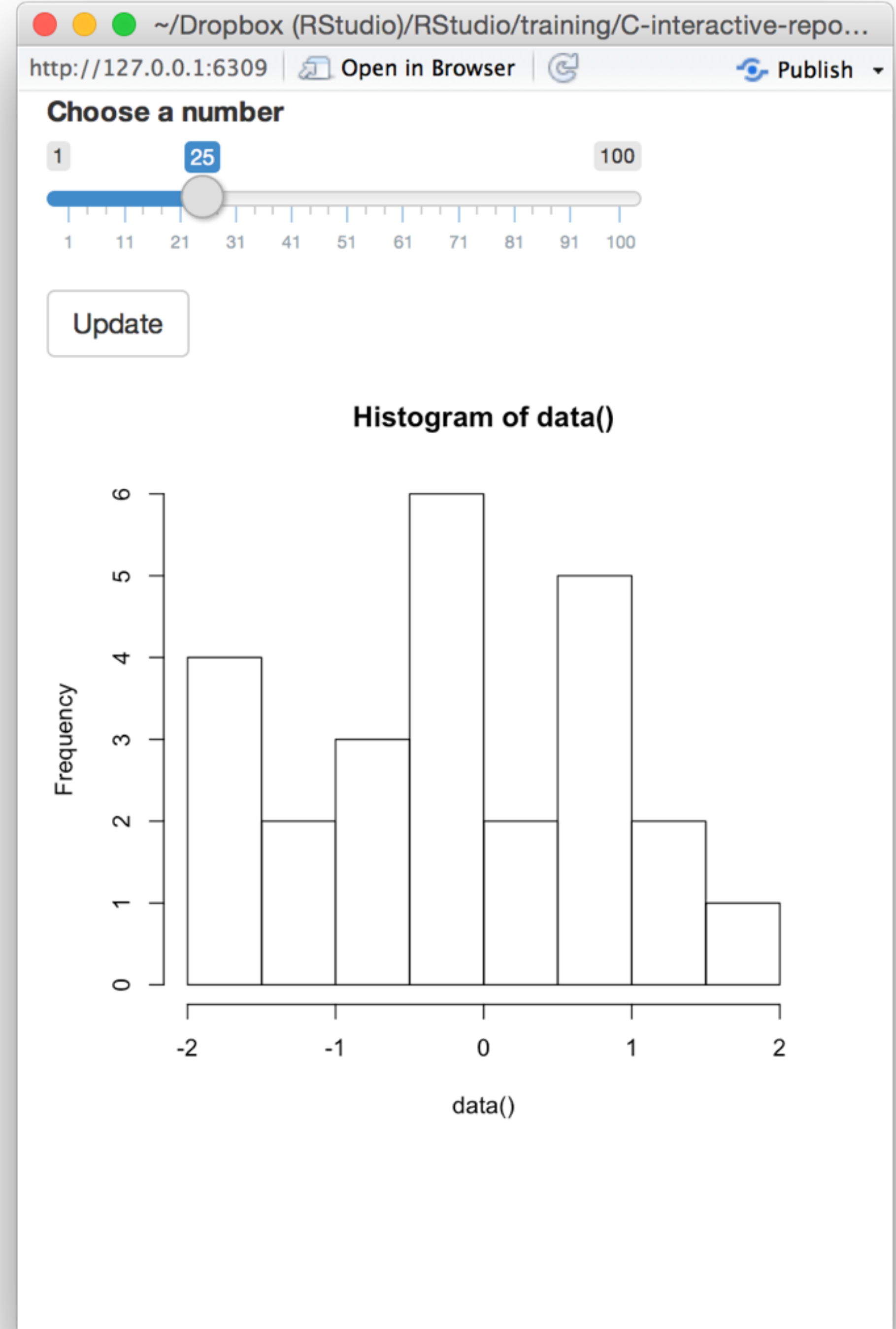
```
# 07-eventReactive
```

```
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)
```



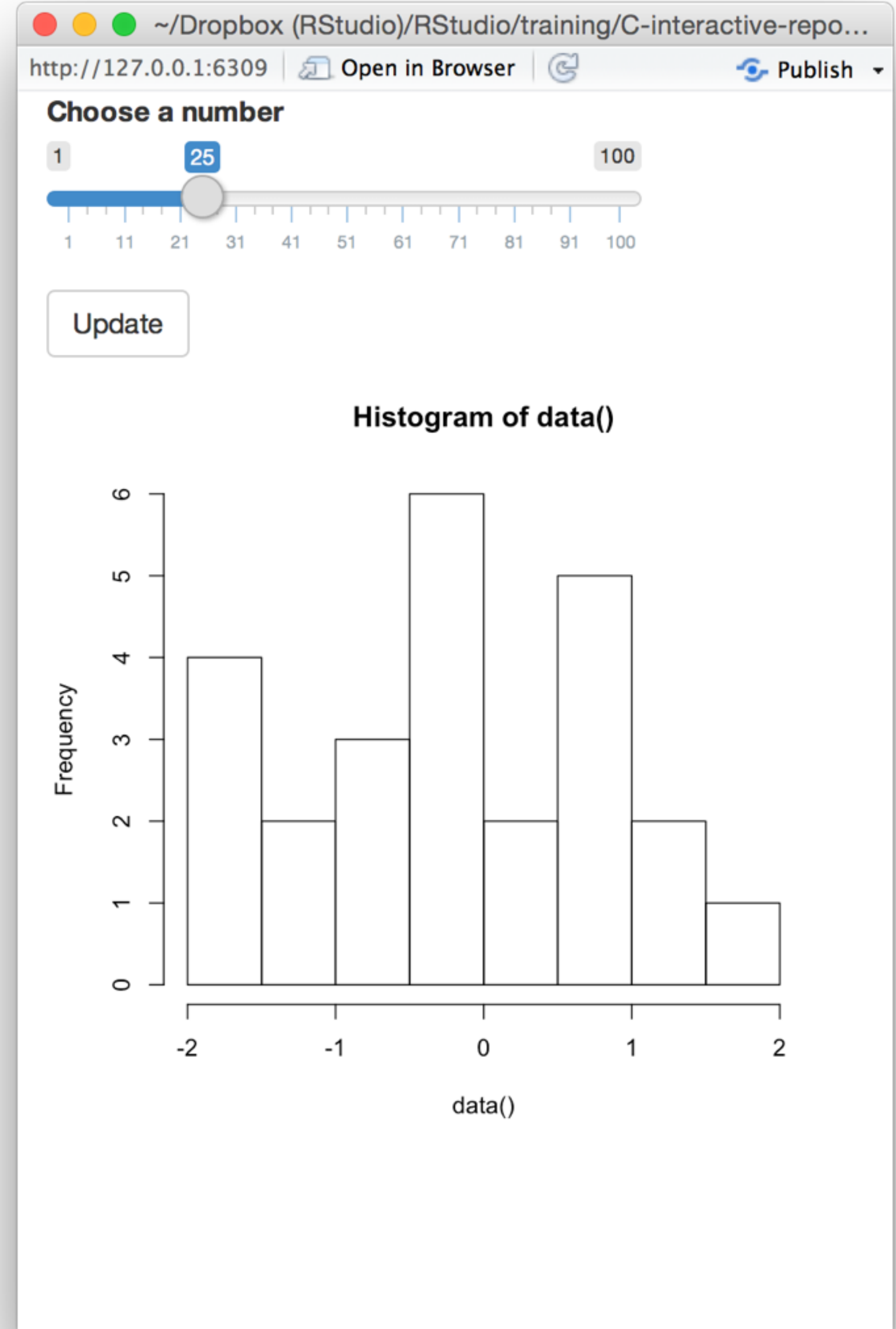
```
# 07-eventReactive
```

```
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)
```

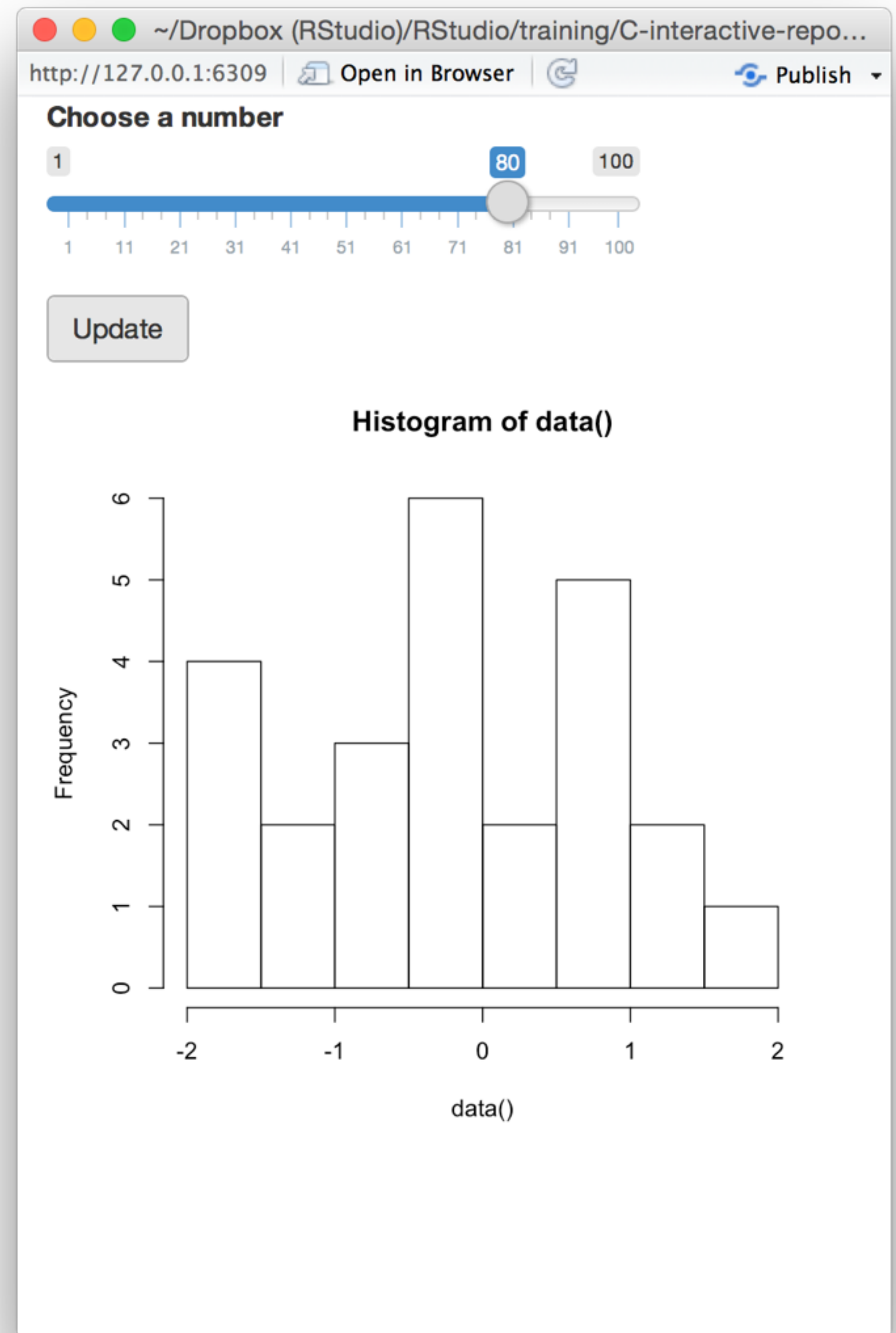


input\$num

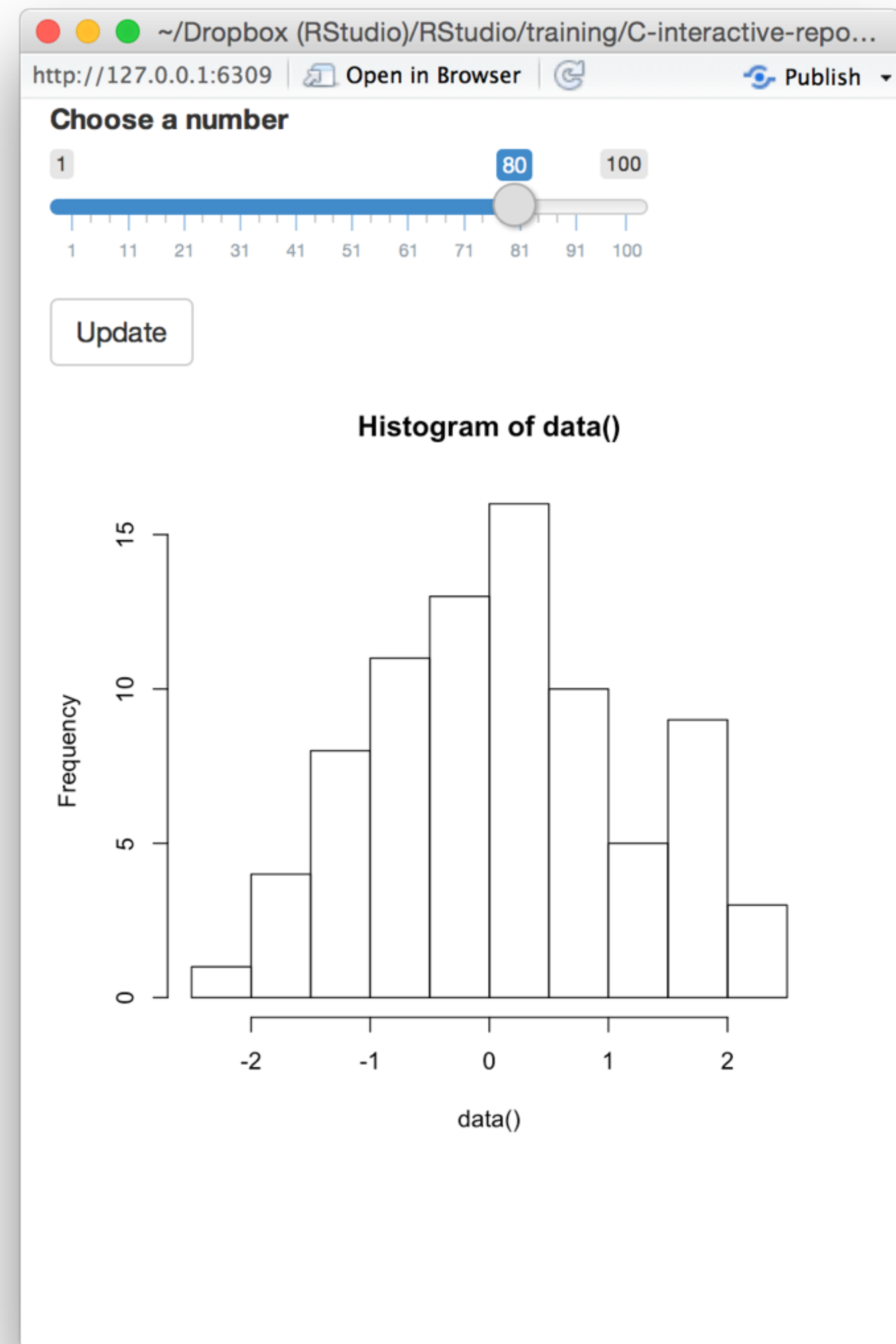
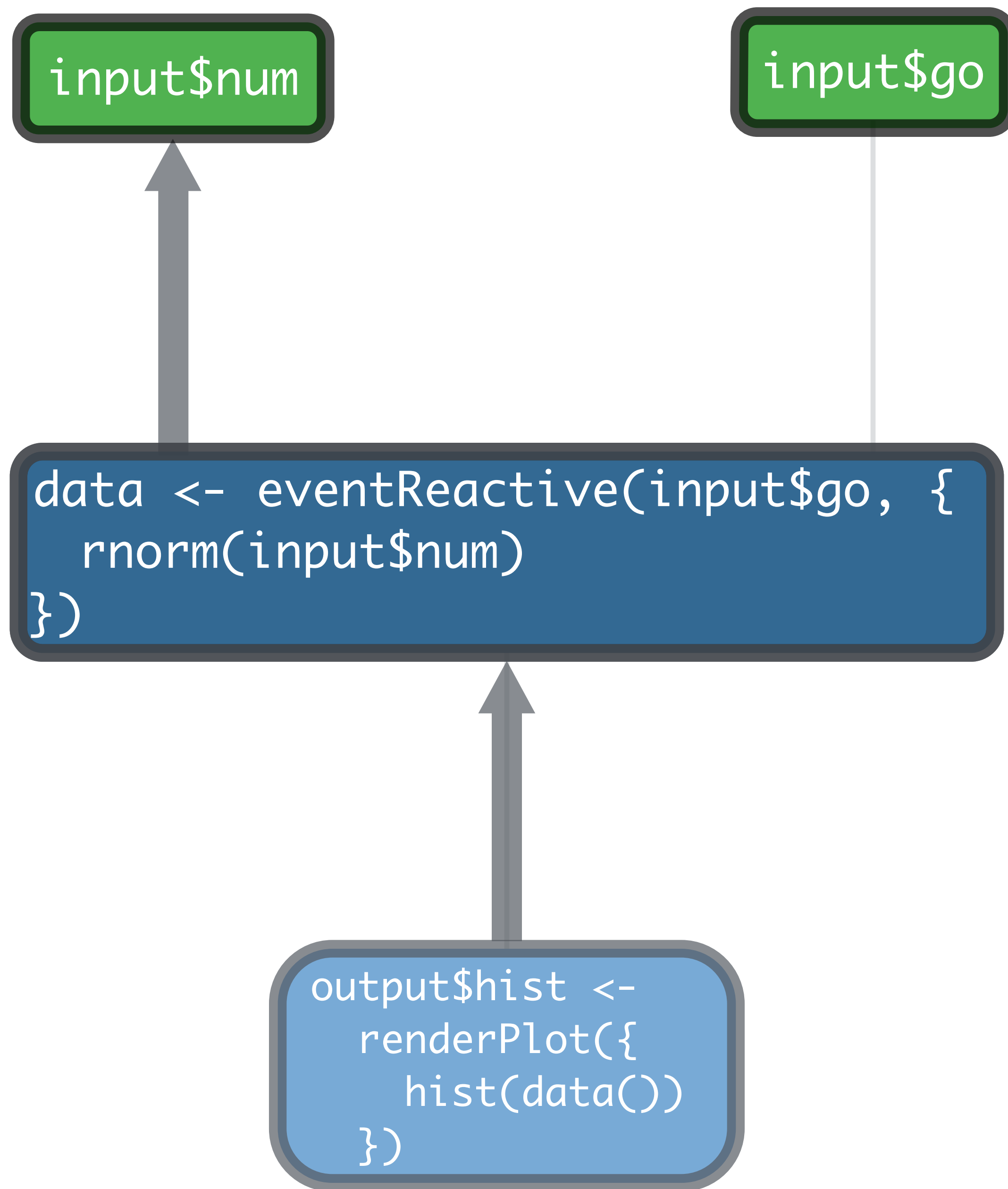
input\$go

```
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) })
```

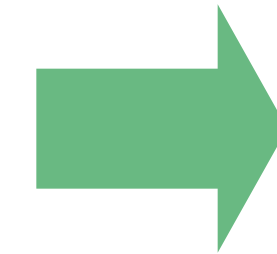
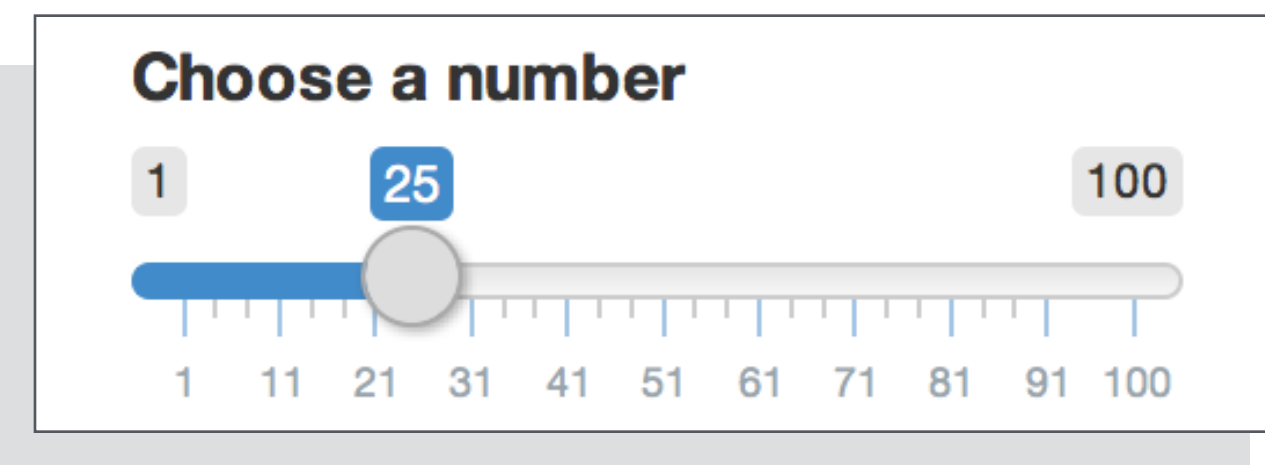
reactive value(s)  
to respond to

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

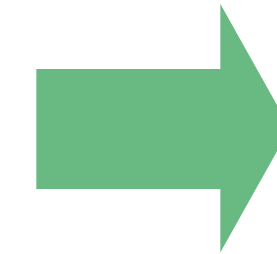
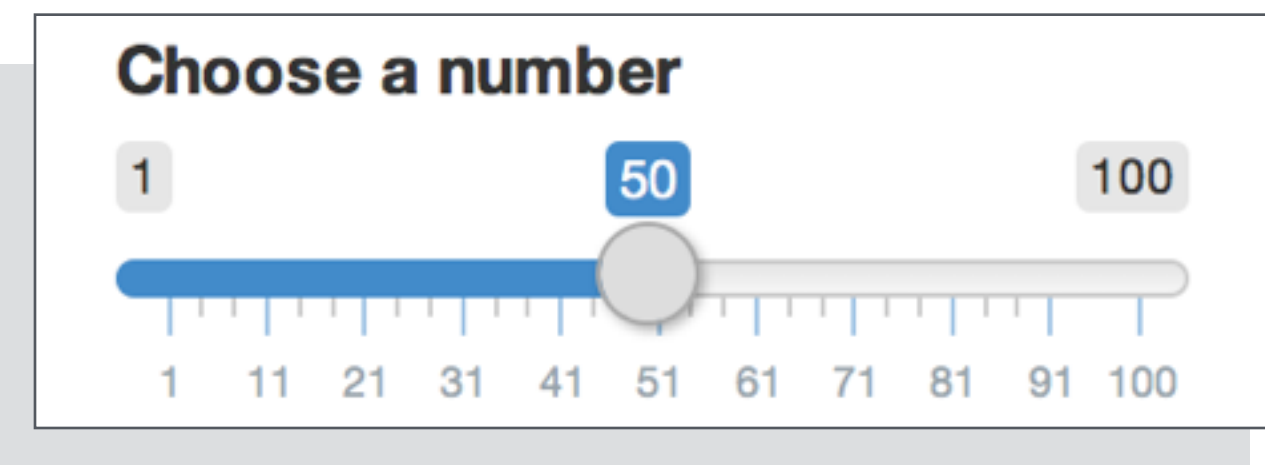
**Manage state**  
**with reactiveValues()**

# Input values

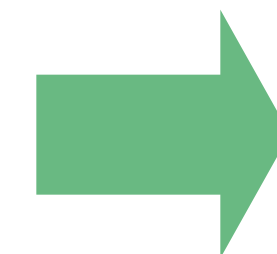
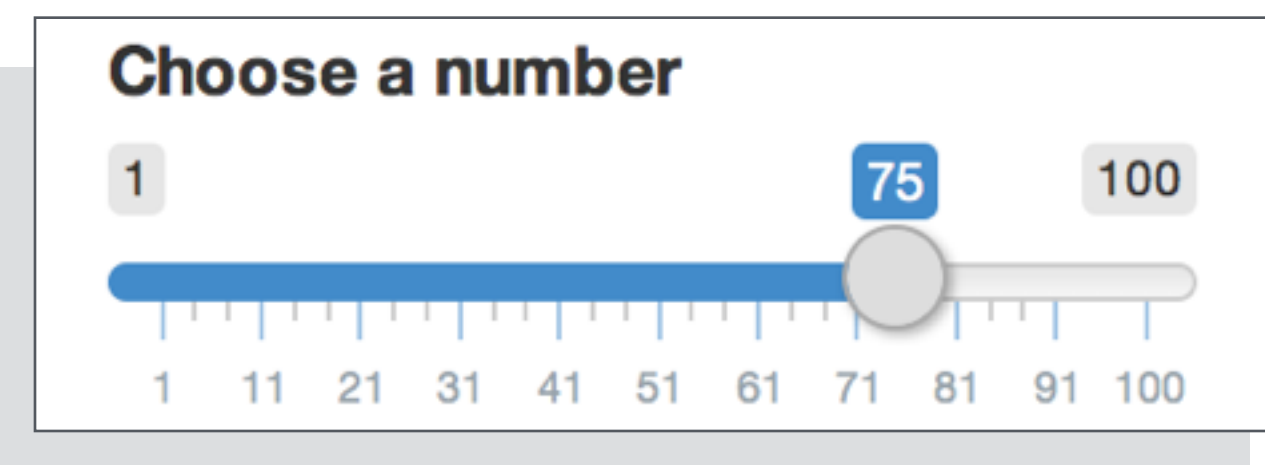
The input value changes whenever a user changes the input.



```
input$num = 25
```



```
input$num = 50
```



```
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) })
```

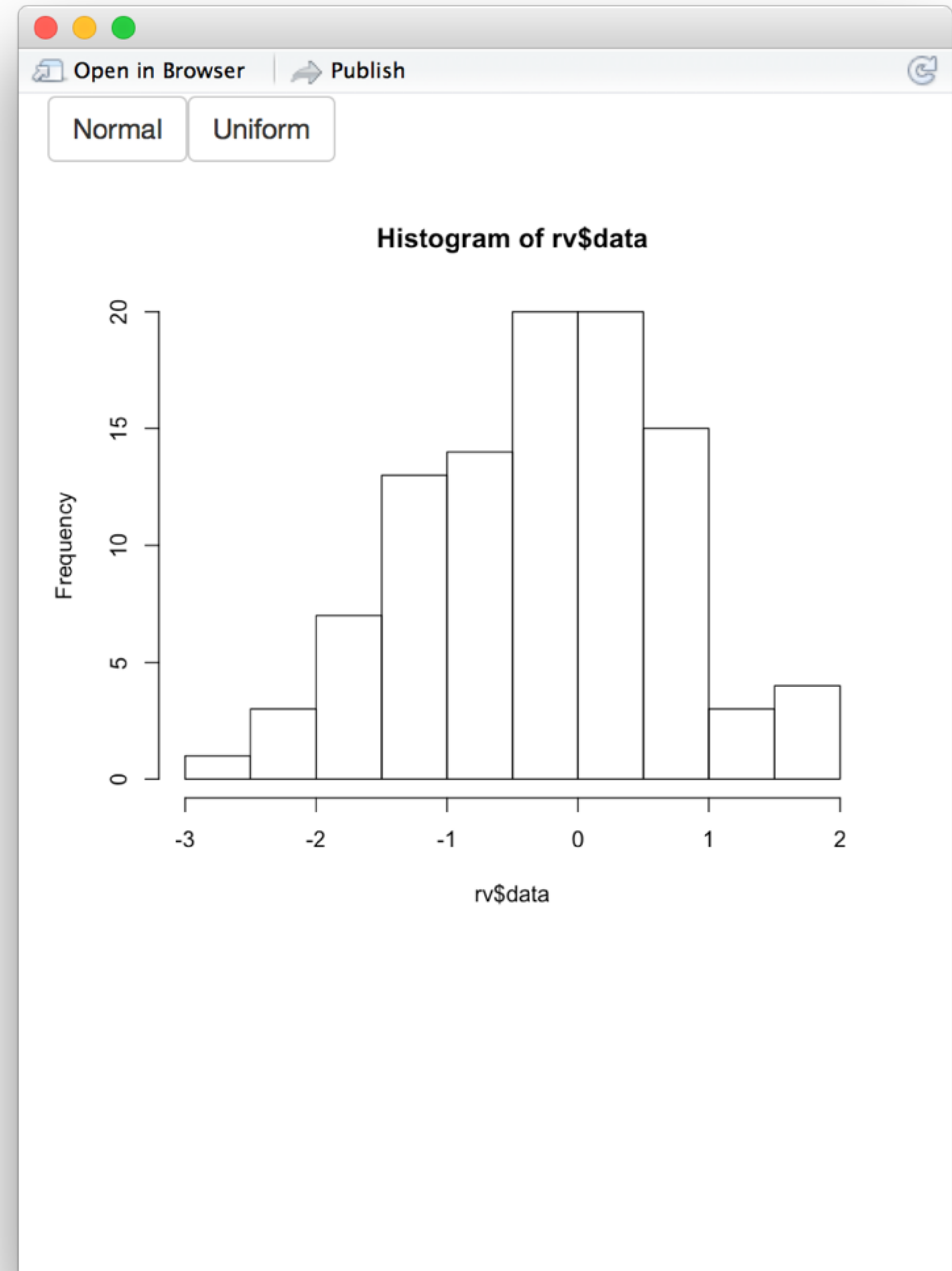
```
  output$hist <- renderPlot({
```

```
    hist(rv$data)
```

```
  })
```

```
}
```

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

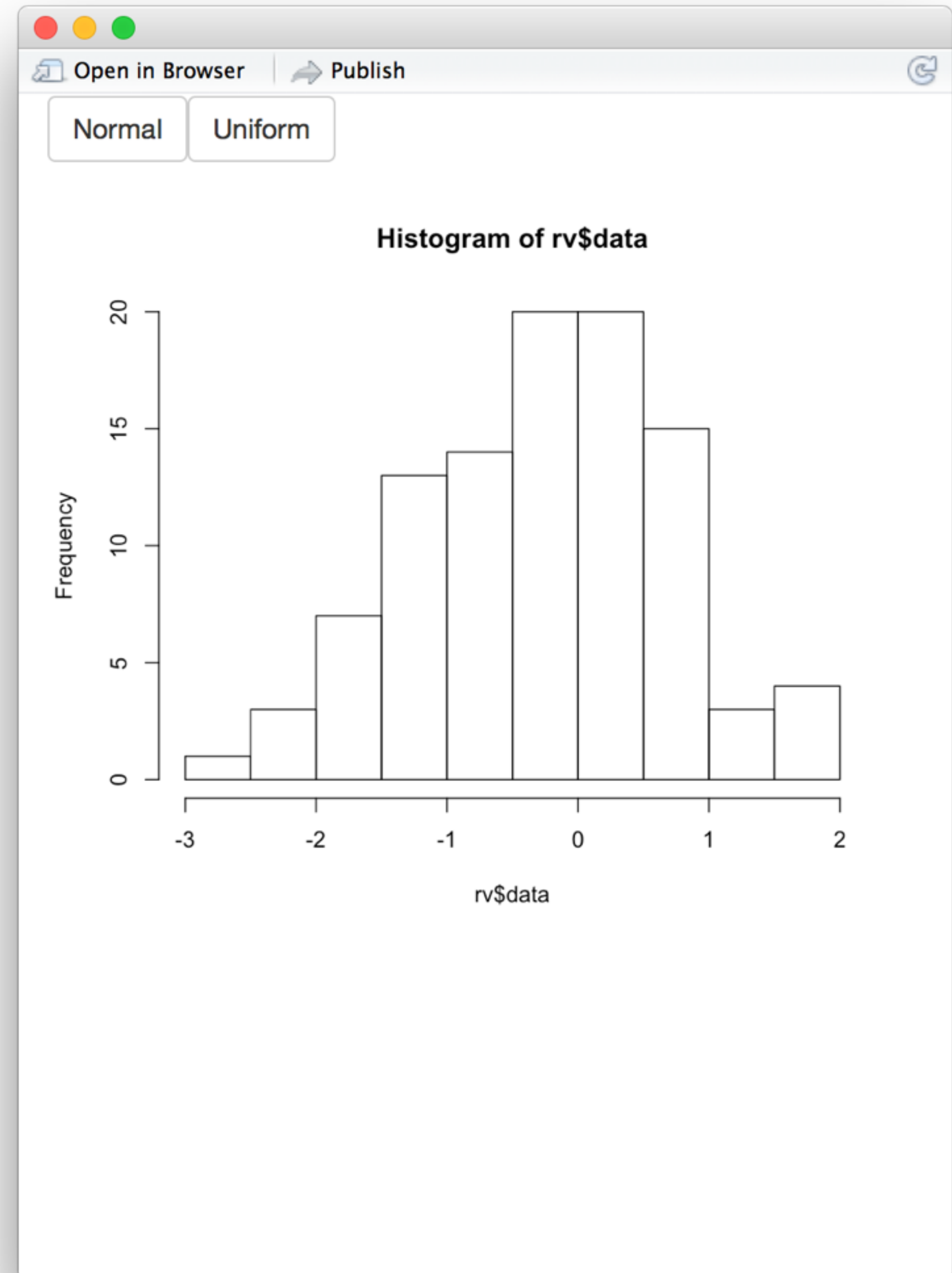


input\$norm

rv\$data  
rnorm(100)

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

input\$unif

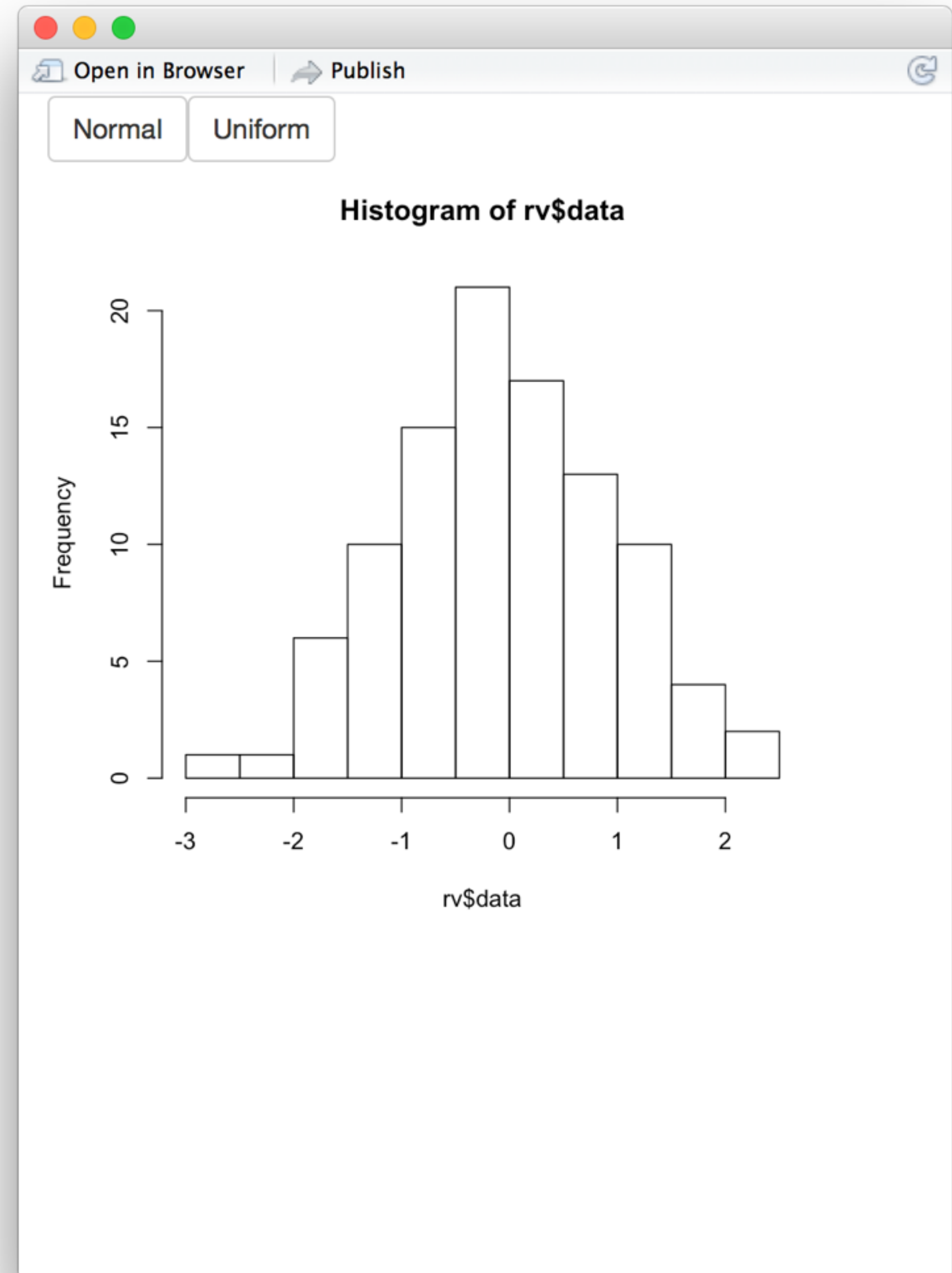


input\$norm

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

input\$unif

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



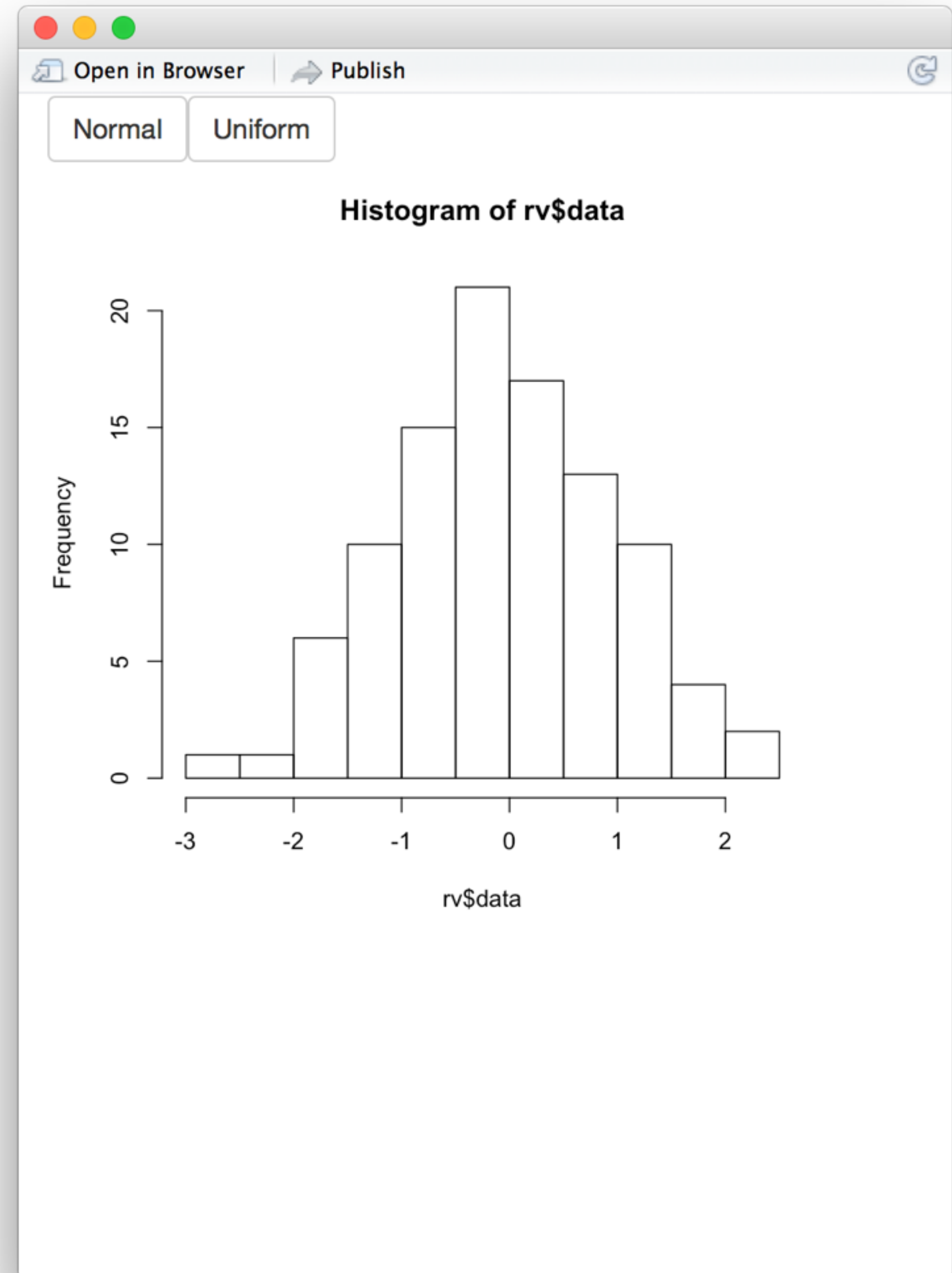


input\$norm

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

input\$unif

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

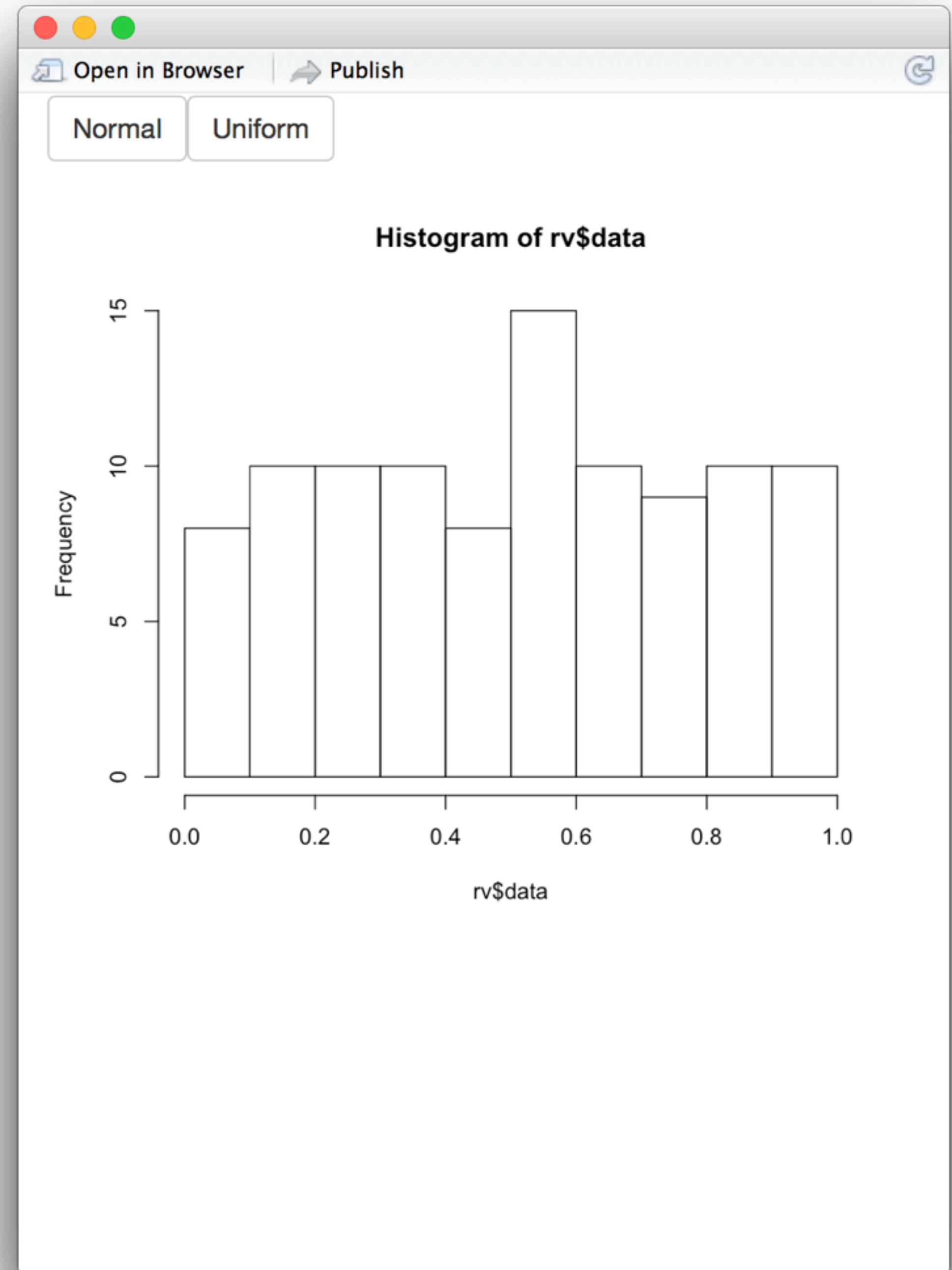


input\$norm

```
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) })
```

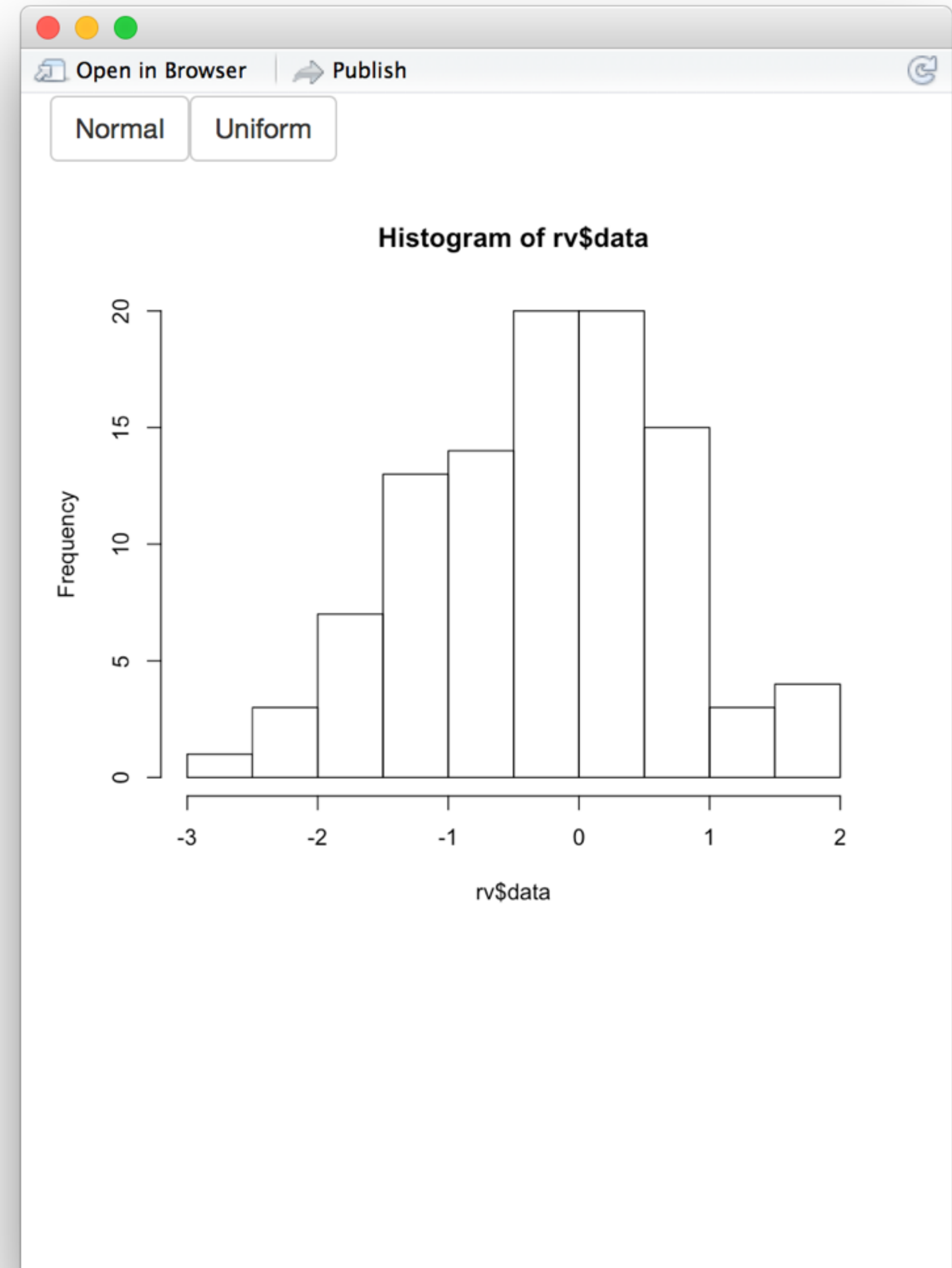
```
  output$hist <- renderPlot({
```

```
    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) })
```

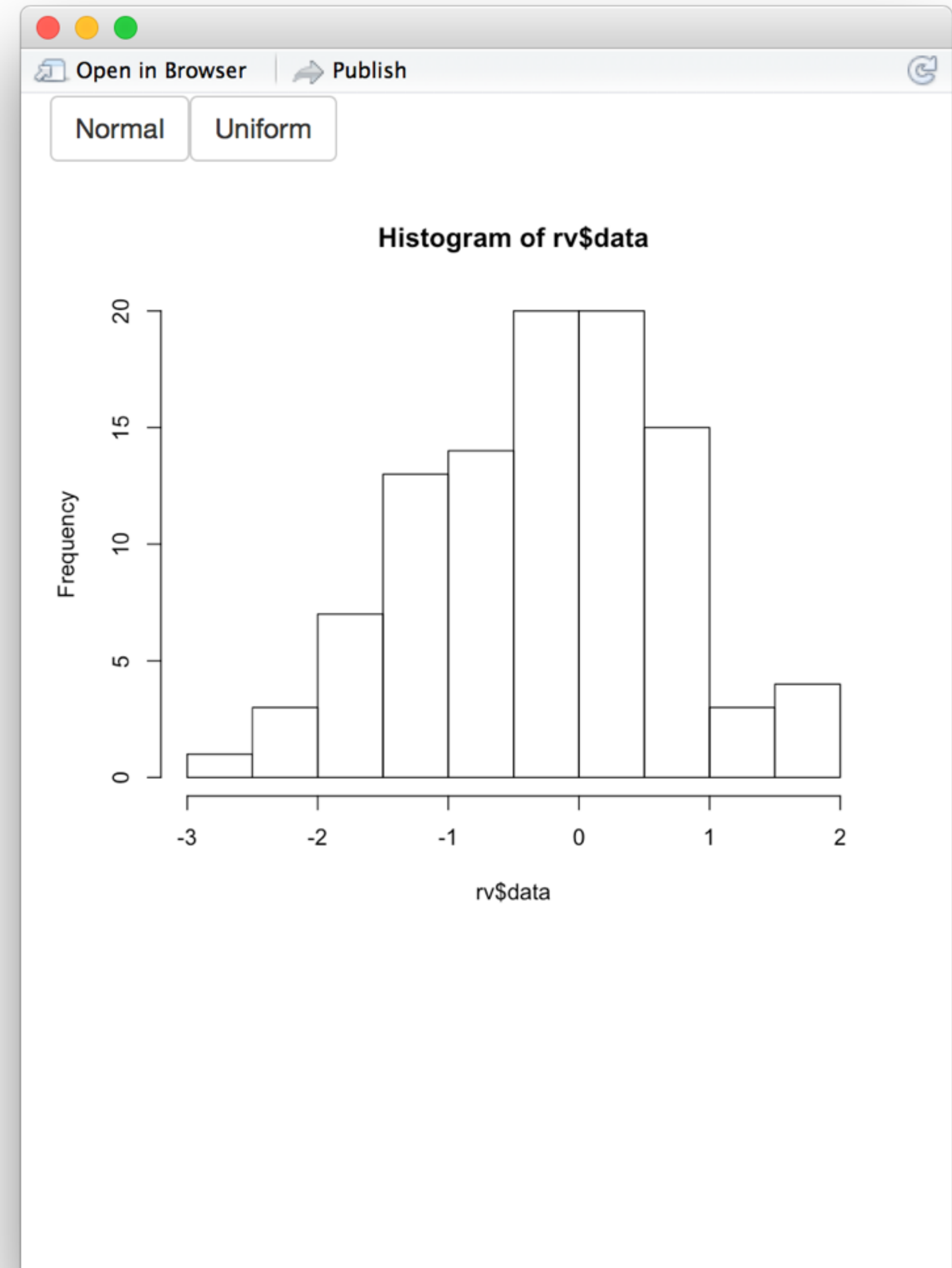
```
  output$hist <- renderPlot({
```

```
    hist(rv$data)
```

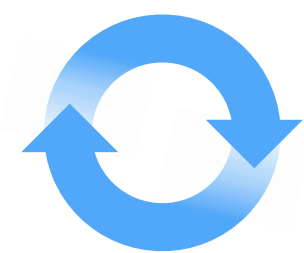
```
  })
```

```
}
```

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



# Recap: reactiveValues()

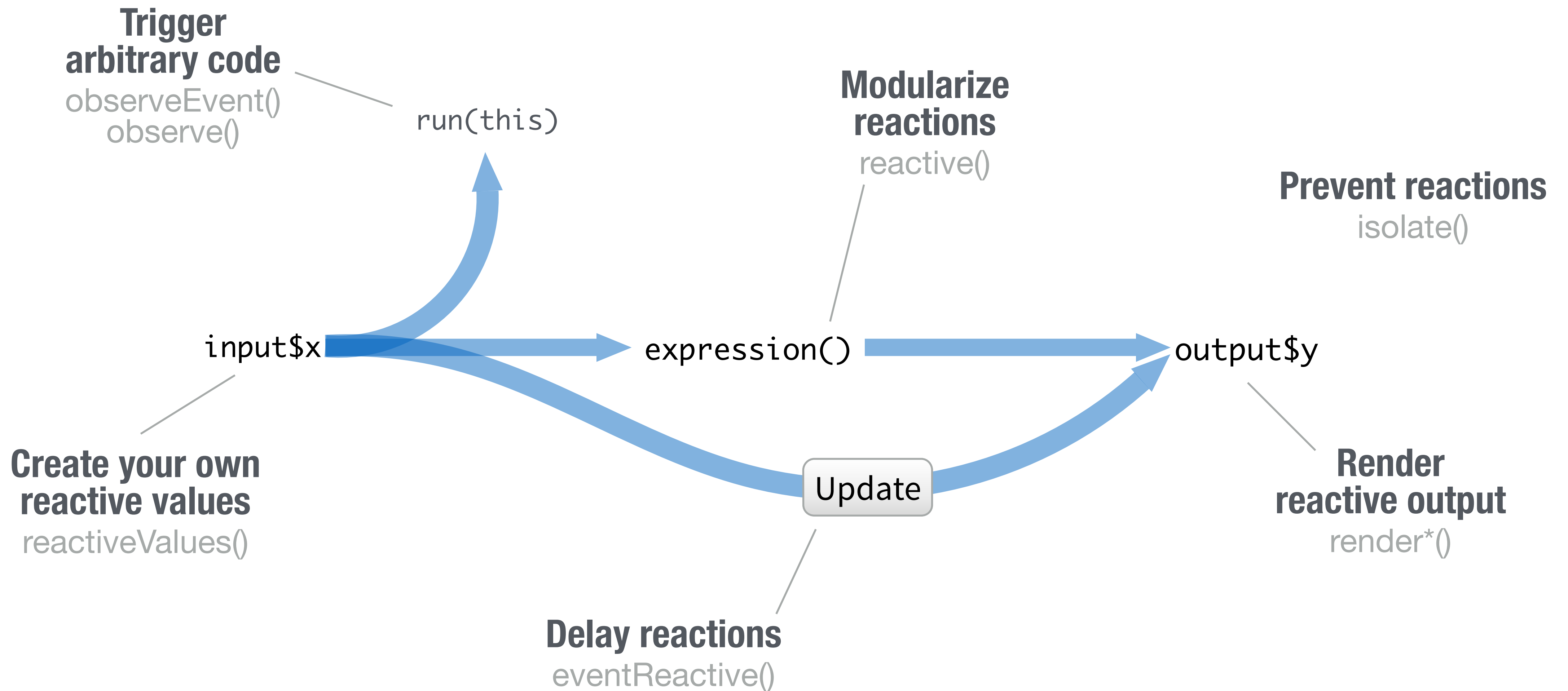


reactiveValues() creates a list of **reactive values**

`rv$data` ←

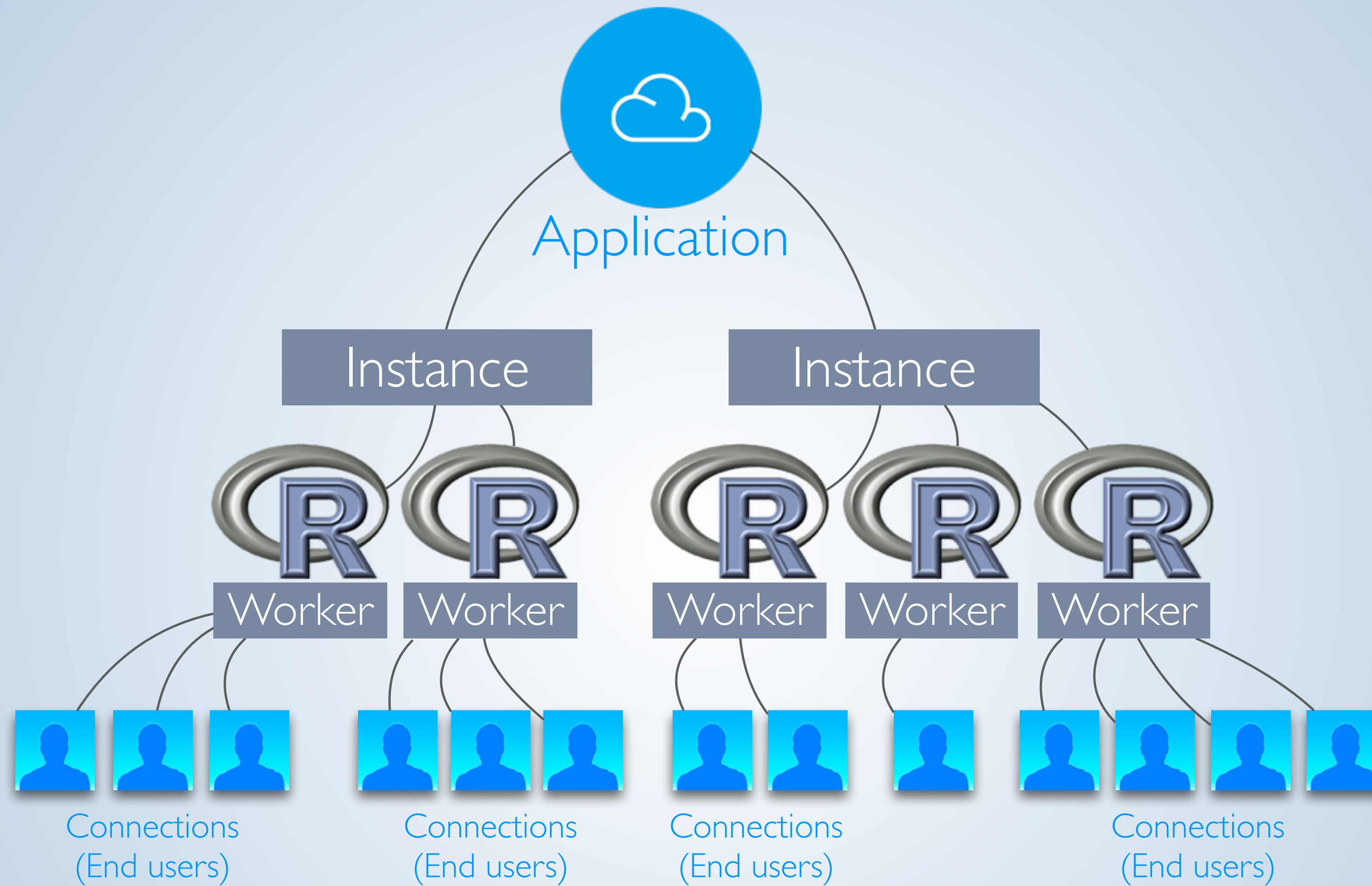
You can manipulate these values (usually with `observeEvent()`)

# 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)
```

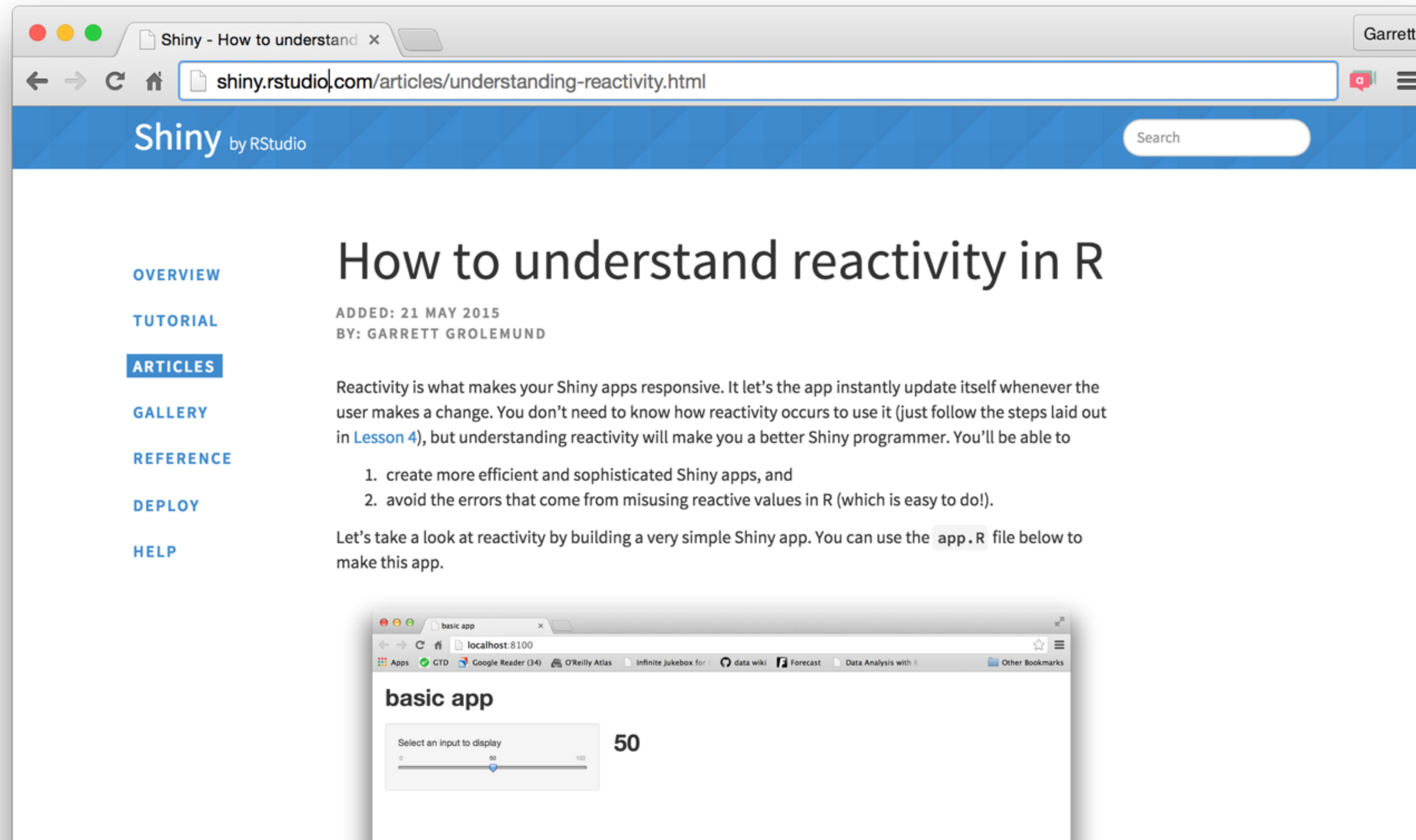
Code outside the server function will be run once per R session (worker)

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 image shows a browser window displaying the Shiny website. The page title is "How to understand reactivity in R" by Garrett Grolemond, dated May 21, 2015. The article text explains that reactivity is what makes Shiny apps responsive and that it allows the app to update itself instantly when the user makes a change. It lists two goals: 1. create more efficient and sophisticated Shiny apps, and 2. avoid the errors that come from misusing reactive values in R. Below the text is a preview of a Shiny app titled "basic app" running on localhost:8100. The app has a slider input labeled "Select an input to display" with a value of 50.

Shiny by RStudio

## How to understand reactivity in R

ADDED: 21 MAY 2015  
BY: GARRETT GROLEMUND

Reactivity is what makes your Shiny apps responsive. It let's 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/](http://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](http://shiny.rstudio.com)

