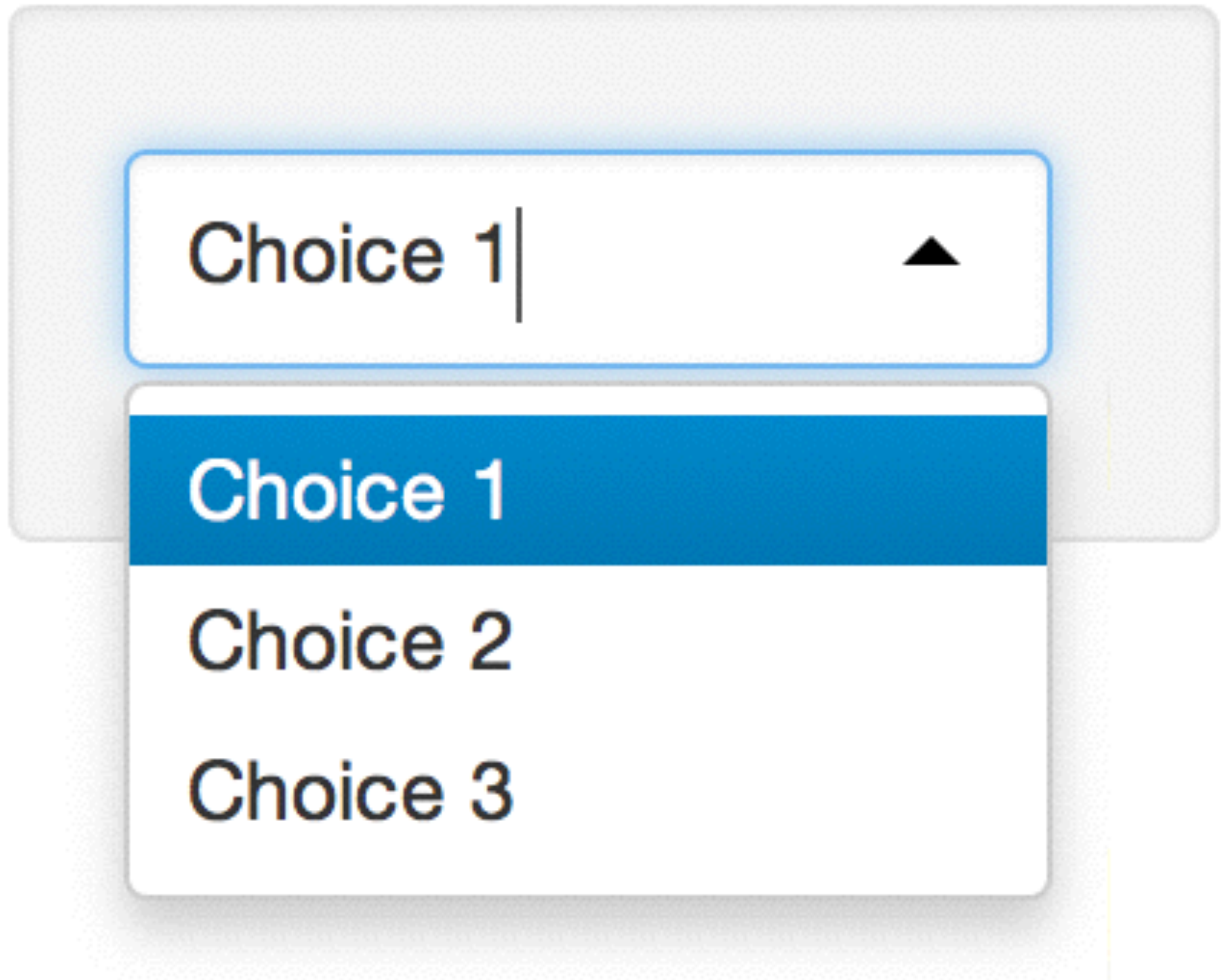
How to start with Shiny, Part 1

How to build a Shiny App



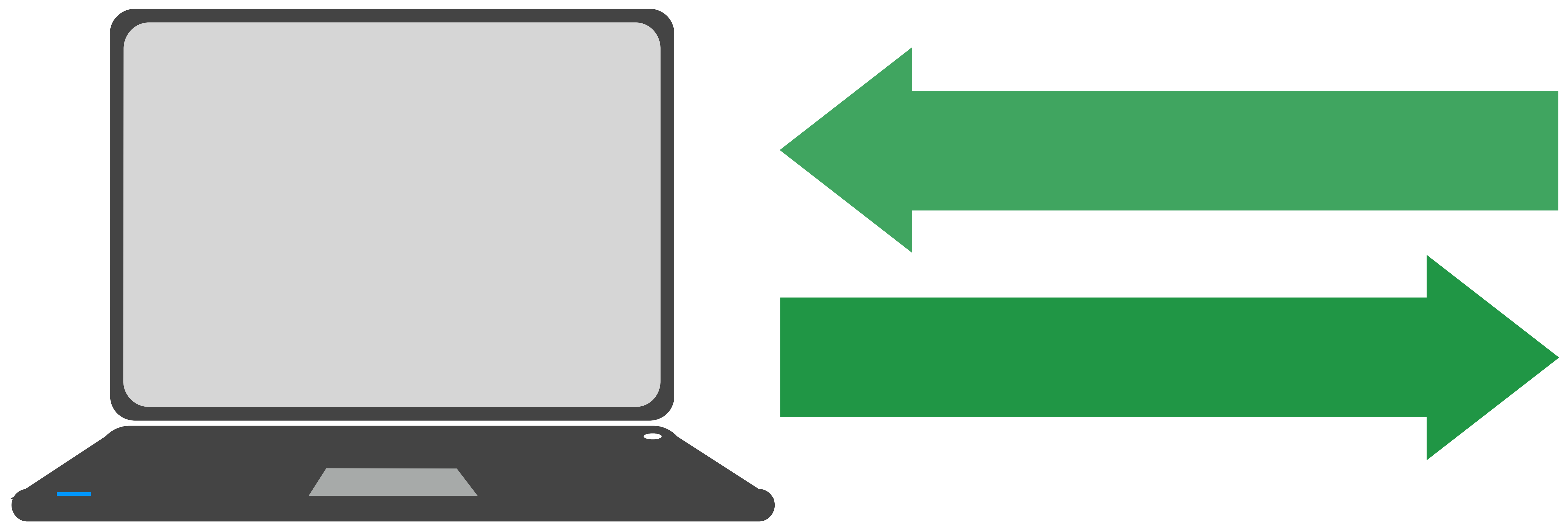
Garrett Grolemund

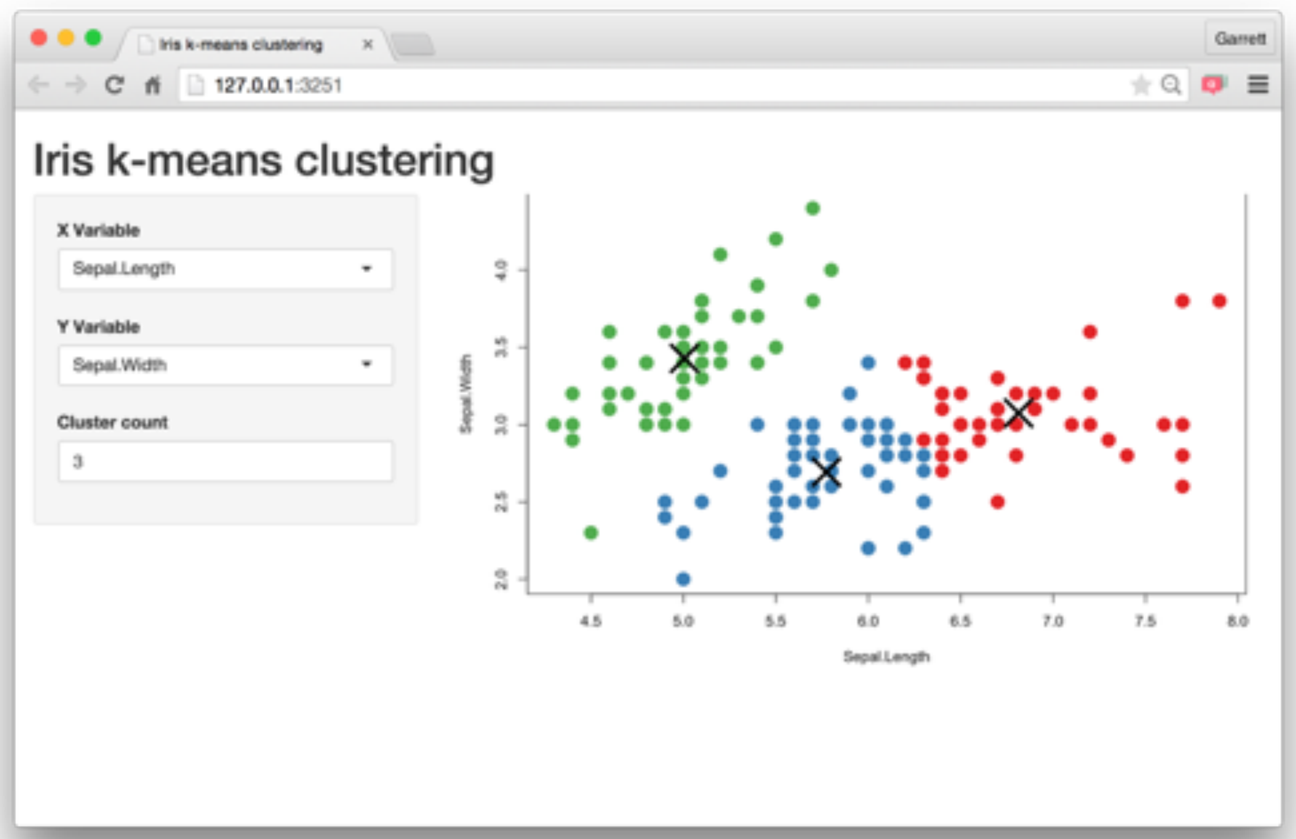
[redacted]

Follow @rstudio

Understand the architecture

Every Shiny app is maintained by a computer running R





Iris k-means clustering

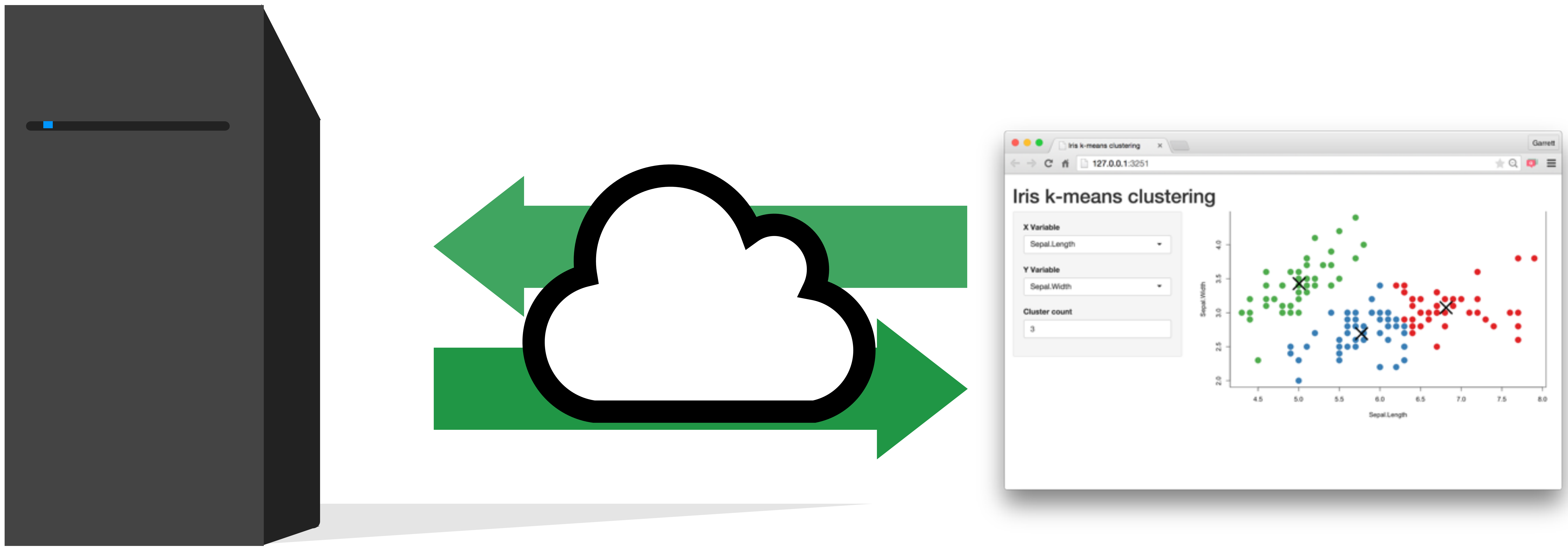
127.0.0.1:3251

X Variable: Sepal.Length

Y Variable: Sepal.Width

Cluster count: 3

Every Shiny app is maintained by a computer running R

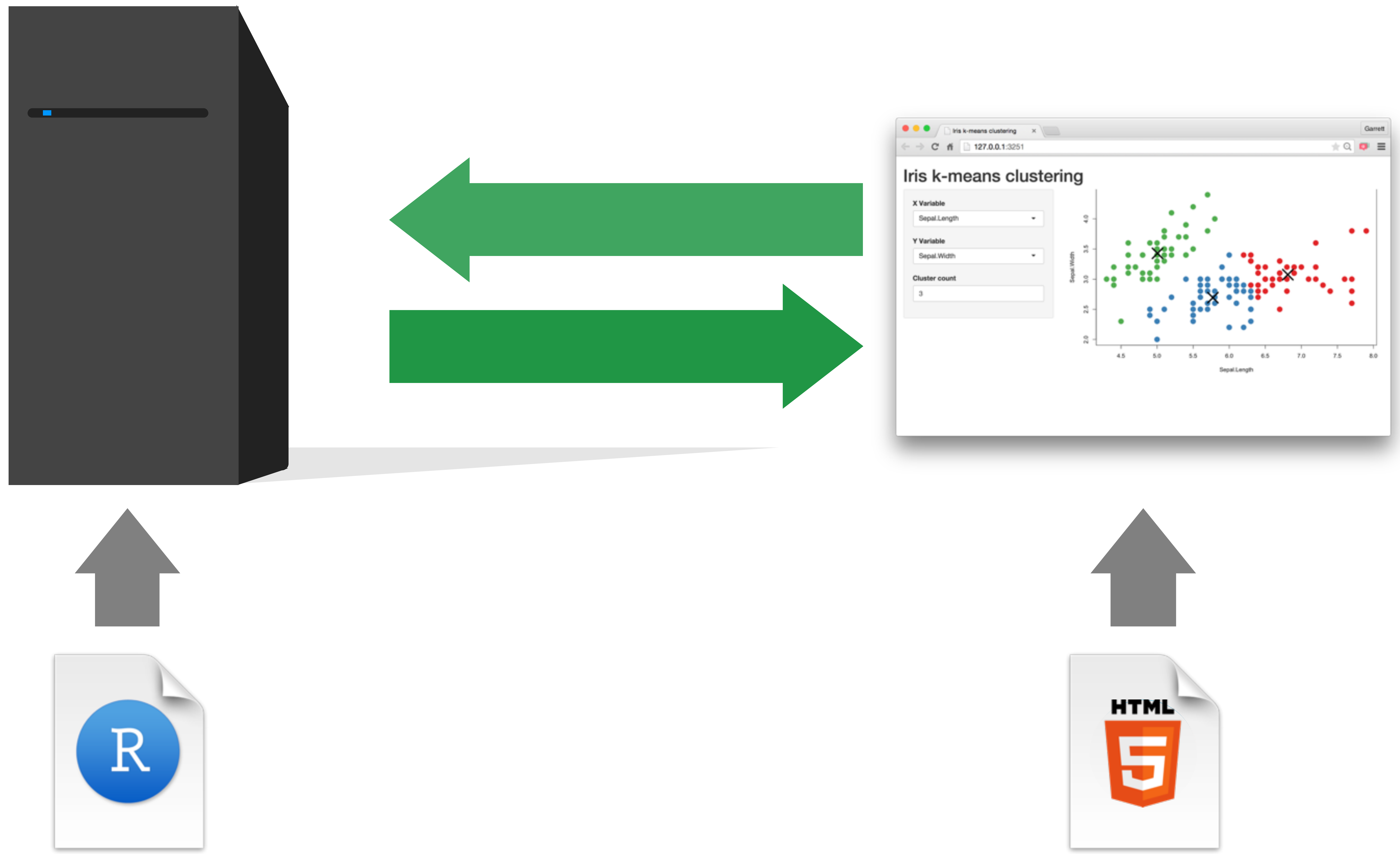

Iris k-means clustering

127.0.0.1:3251

X Variable: Sepal.Length

Y Variable: Sepal.Width

Cluster count: 3



Server Instructions

User Interface (UI)

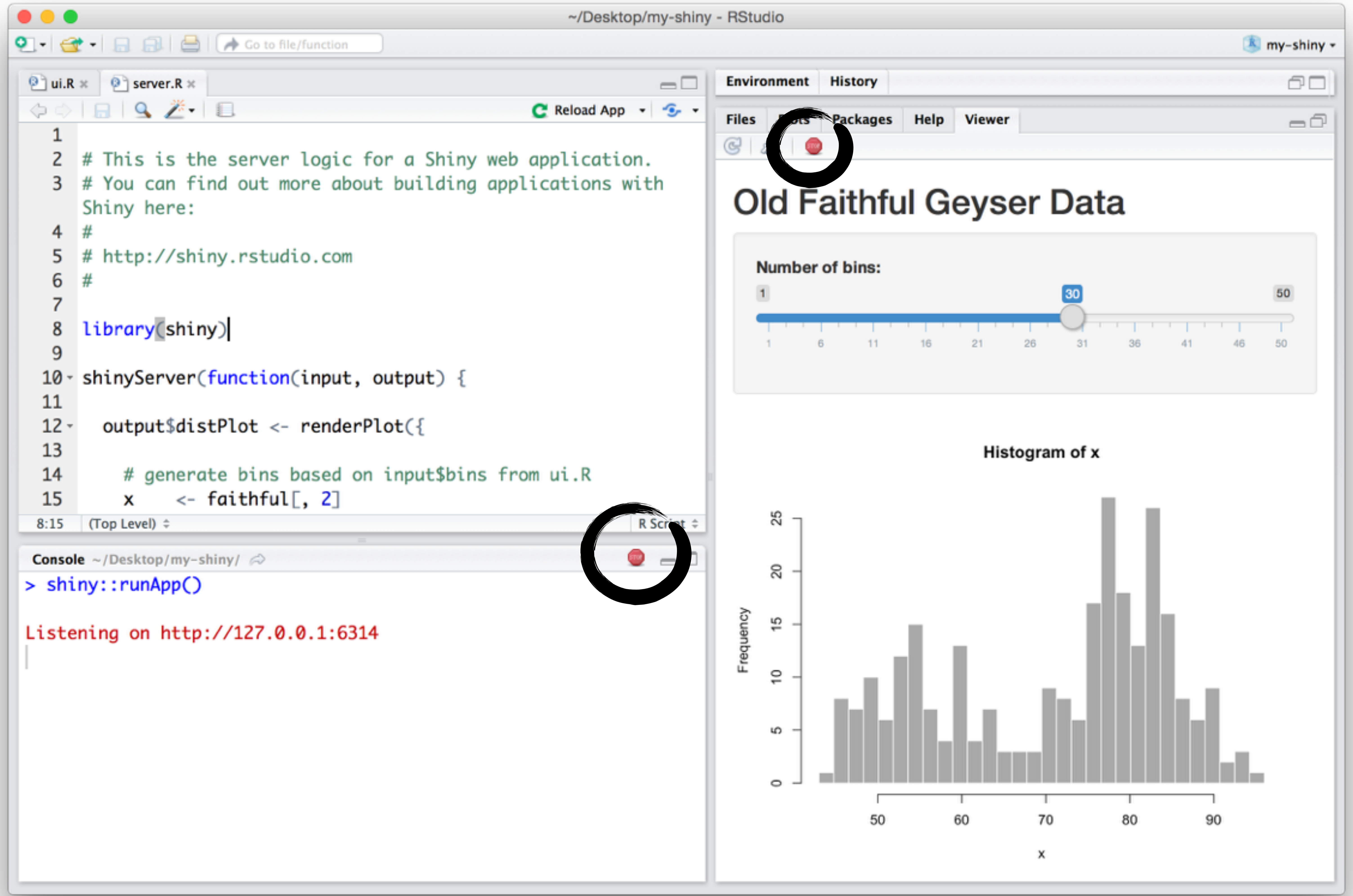
Use the template

App template

The shortest viable shiny app

```r
library(shiny)
ui <- fluidPage()

server <- function(input, output) {}

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

Close an app



```r

# This is the server logic for a Shiny web application.
# You can find out more about building applications with Shiny here:
#
# http://shiny.rstudio.com
#

library(shiny)

shinyServer(function(input, output) {

  output$distPlot <- renderPlot({

    # generate bins based on input$bins from ui.R
    x    <- faithful[, 2]
```

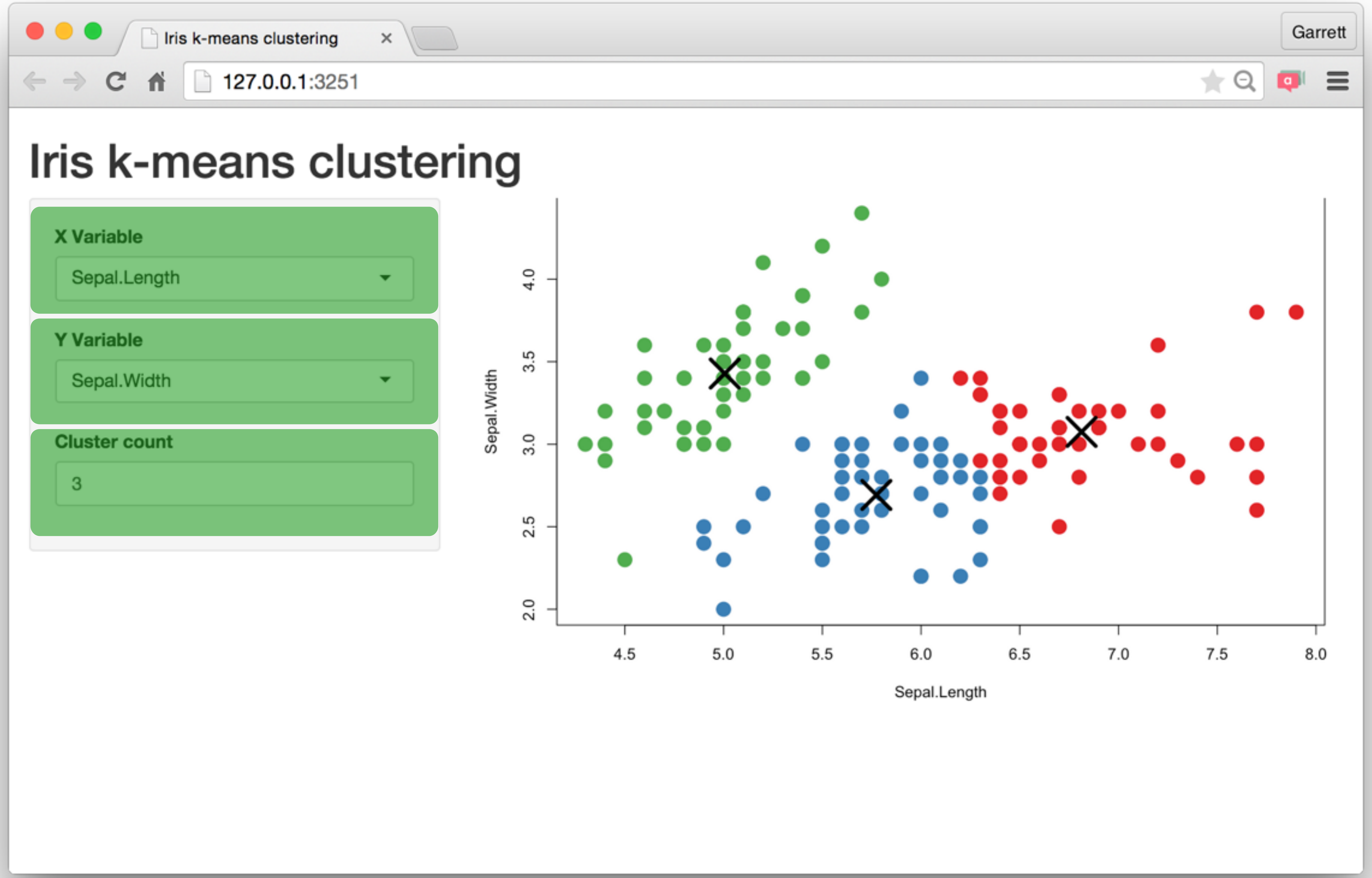
```
> shiny::runApp()

Listening on http://127.0.0.1:6314
```

Add elements to your app as arguments to `fluidPage()`

```
library(shiny)
ui <- fluidPage("Hello World")

server <- function(input, output) {}

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

Build your app around Inputs and Outputs

Build your app around inputs and outputs



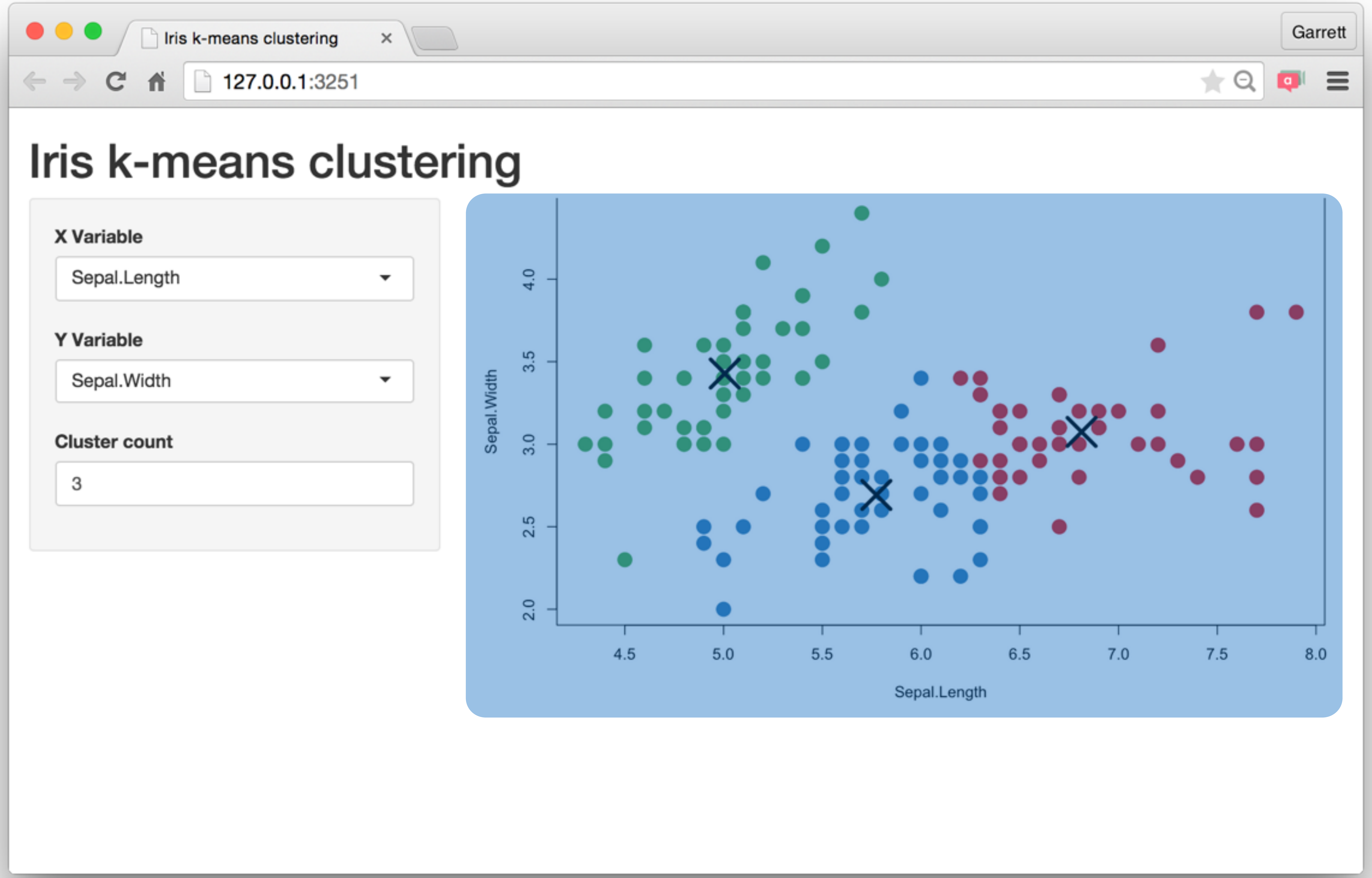
Iris k-means clustering

X Variable
Sepal.Length

Y Variable
Sepal.Width

Cluster count
3

Build your app around inputs and outputs



Iris k-means clustering

X Variable: Sepal.Length

Y Variable: Sepal.Width

Cluster count: 3

Add elements to your app as arguments to `fluidPage()`

```
ui <- fluidPage(
  # *Input() functions,
  # *Output() functions
)
```

Inputs

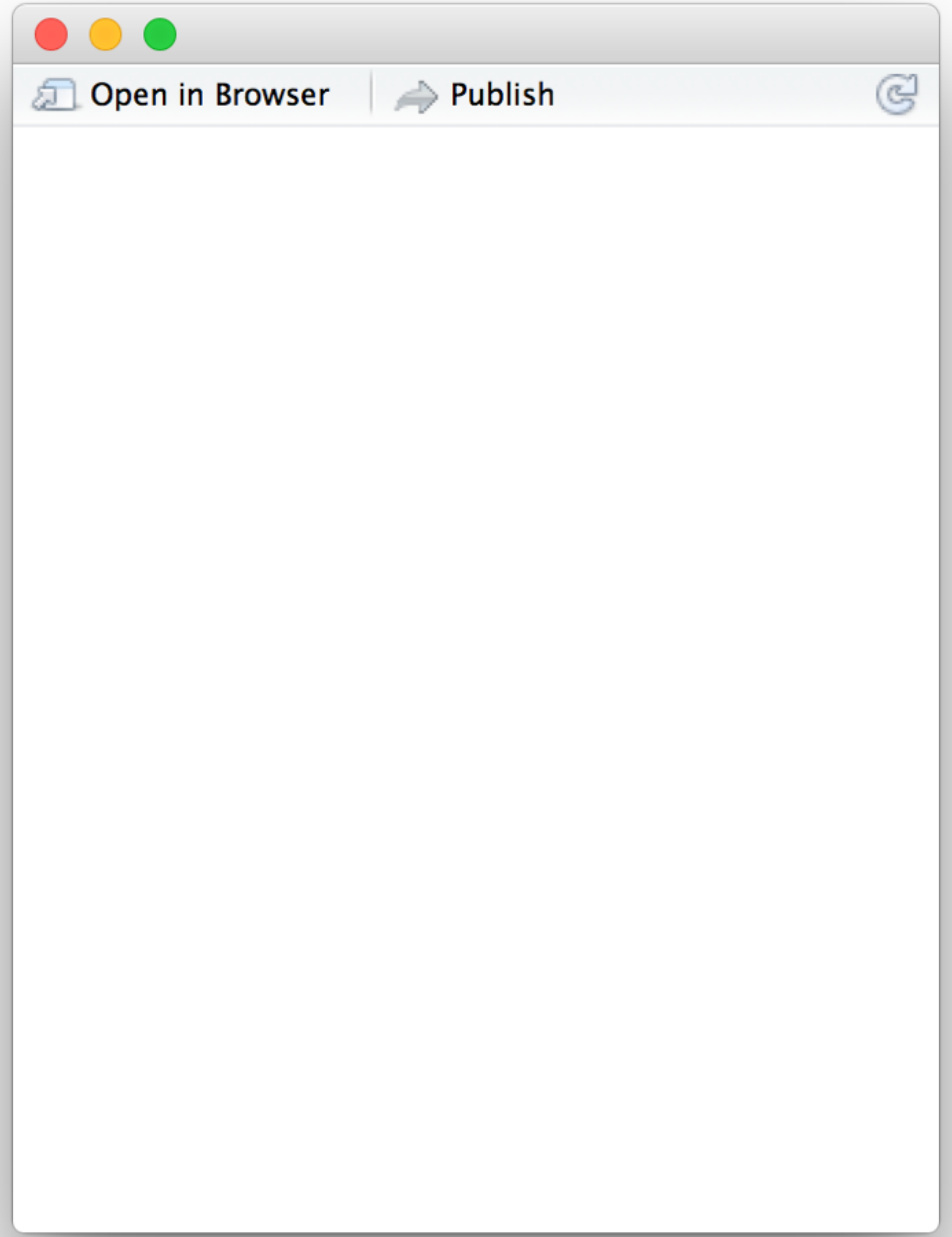
Create an input with an ***Input()** function.

```
sliderInput(inputId = "num",
  label = "Choose a number",
  value = 25, min = 1, max = 100)
```

```
<div class="form-group shiny-input-container">
  <label class="control-label" for="num">Choose a number</label>
  <input class="js-range-slider" id="num" data-min="1" data-max="100"
   data-from="25" data-step="1" data-grid="true" data-grid-num="9.9"
   data-grid-snap="false" data-prettify-separator="," data-keyboard="true"
   data-keyboard-step="1.01010101010101"/>
</div>
```

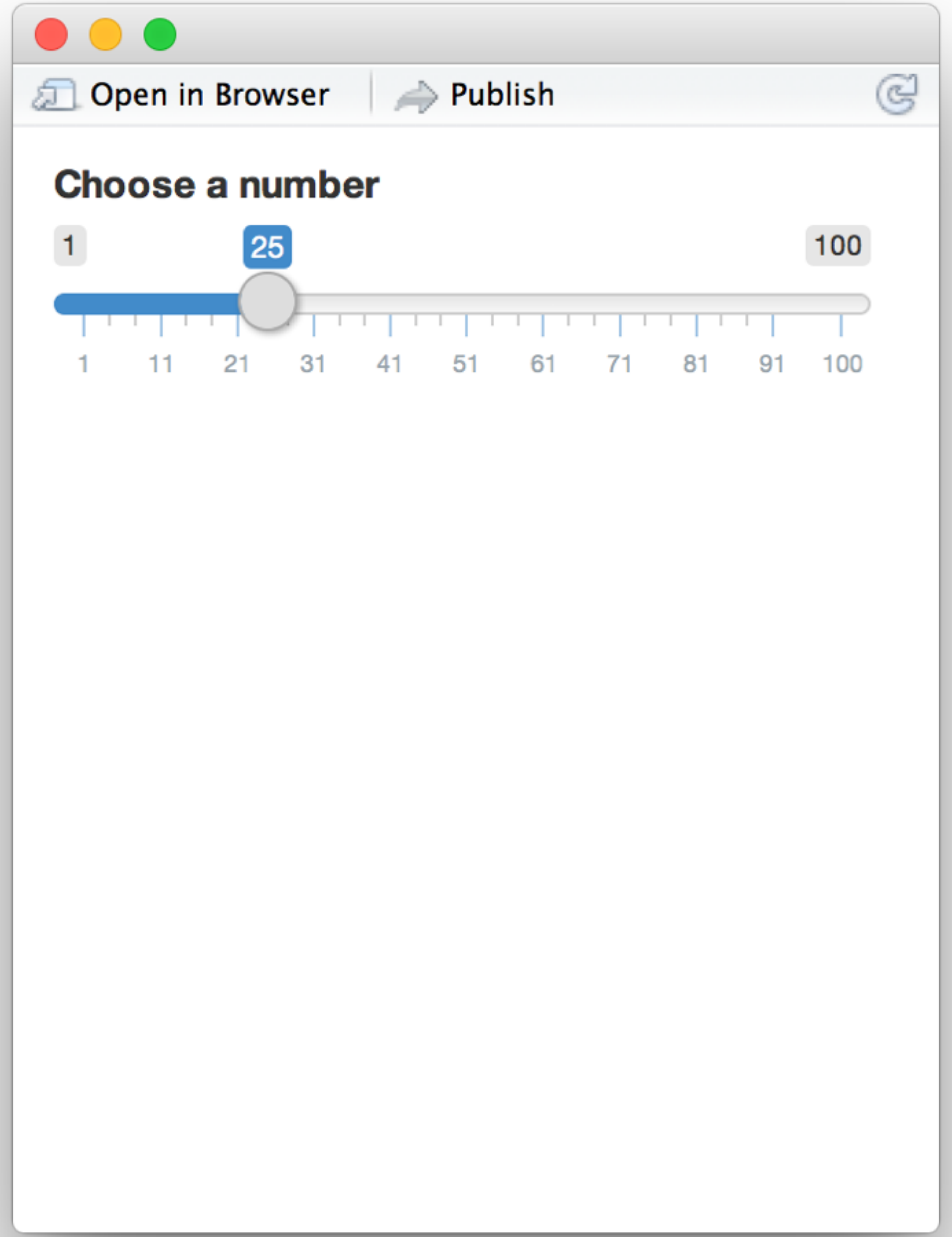
Create an input with an input function.

```
library(shiny)
ui <- fluidPage(




)

server <- function(input, output) {}

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

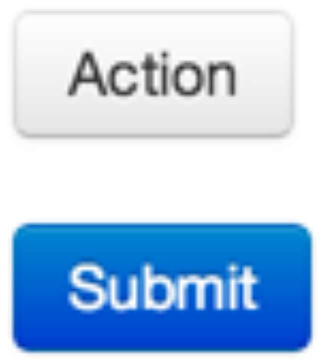


Create an input with an input function.

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

server <- function(input, output) {}

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

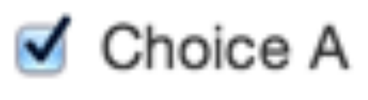


Buttons



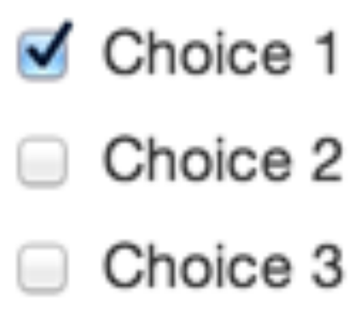
`actionButton()`
`submitButton()`

Single checkbox



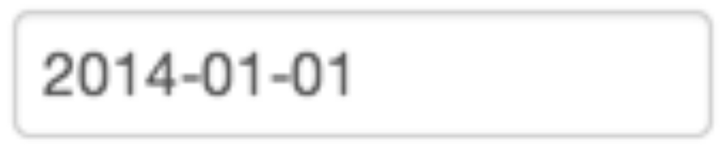
`checkboxInput()`

Checkbox group



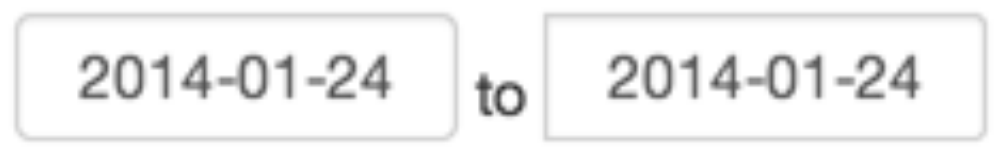
`checkboxGroupInput()`

Date input



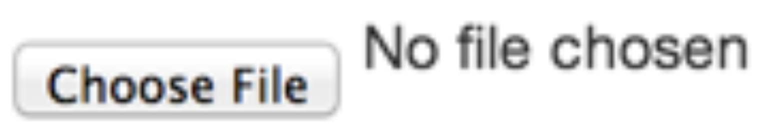
`dateInput()`

Date range



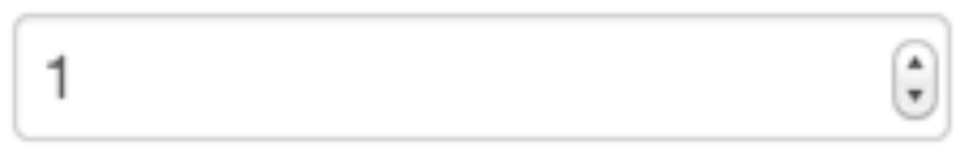
`dateRangeInput()`

File input



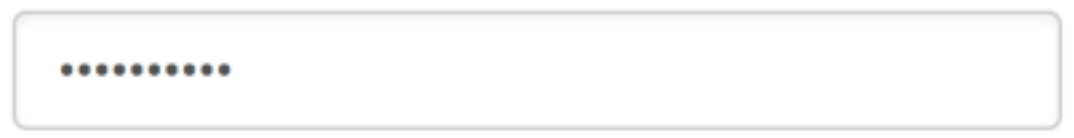
`fileInput()`

Numeric input



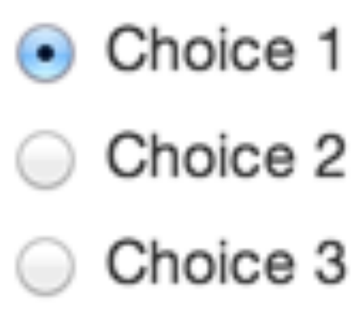
`numericInput()`

Password Input



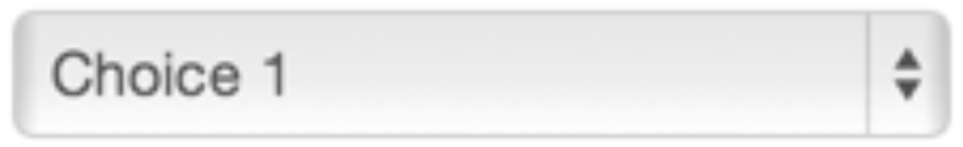
`passwordInput()`

Radio buttons



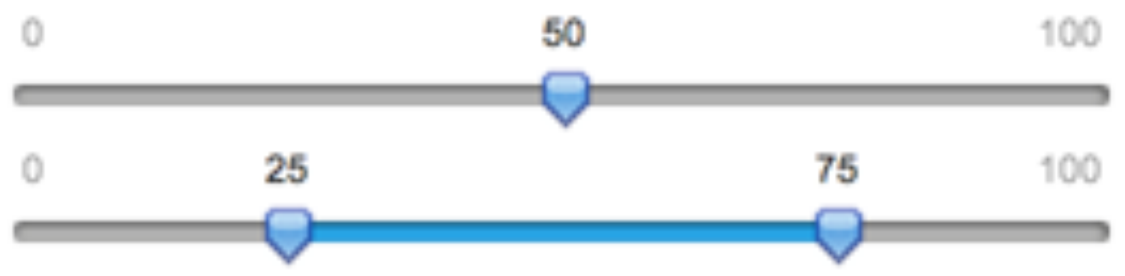
`radioButtons()`

Select box



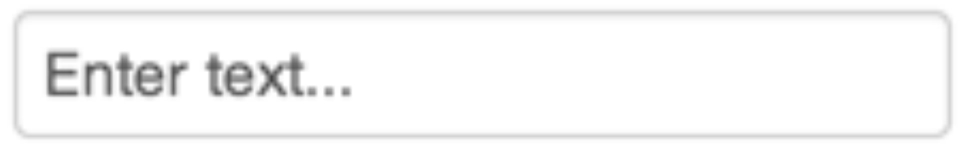
`selectInput()`

Sliders



`sliderInput()`

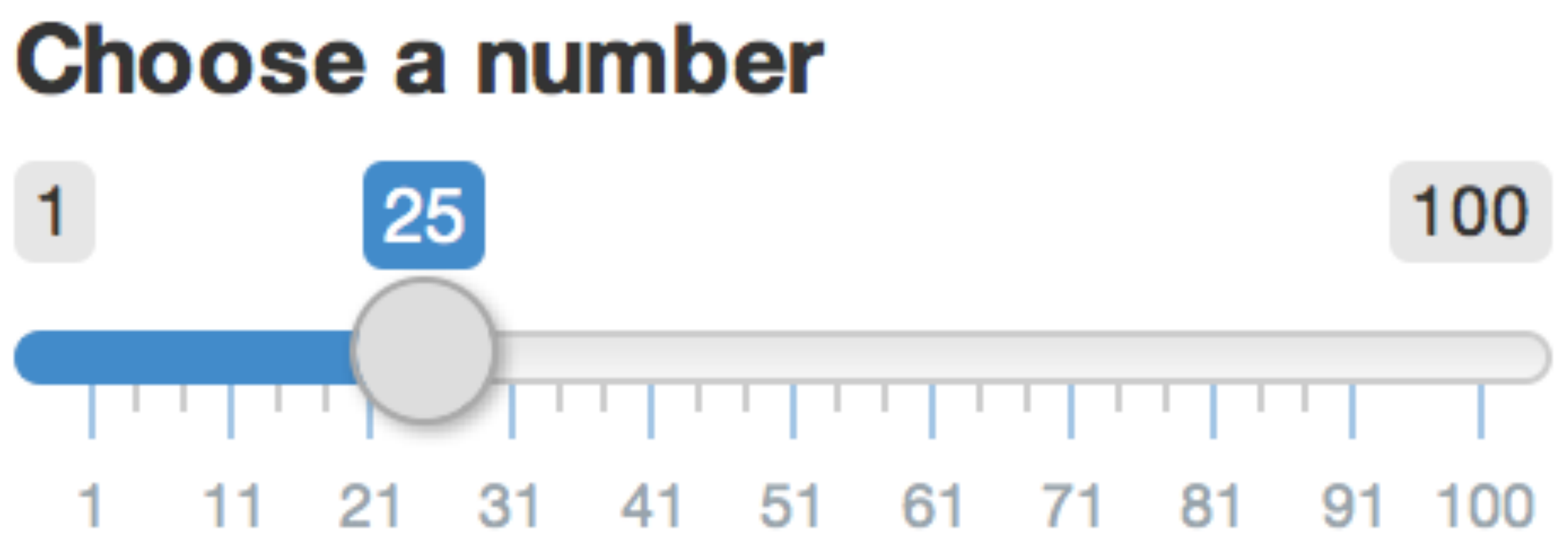
Text input



`textInput()`

RStudio

Syntax



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

input name
(for internal use)

Notice:
Id not ID

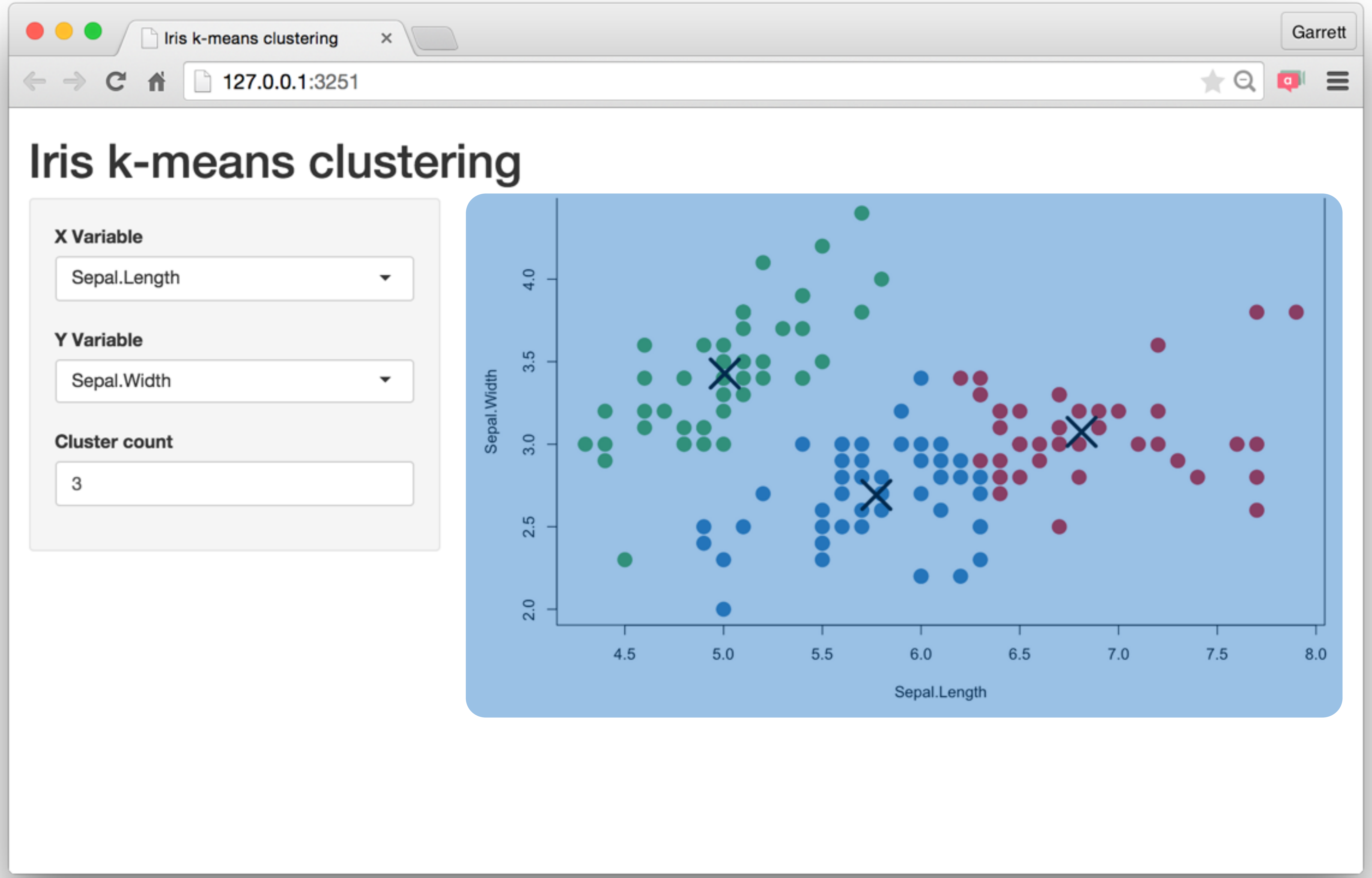
label to
display

input specific
arguments

```
?sliderInput
```

Outputs

Build your app around inputs and outputs



Iris k-means clustering

X Variable

Sepal.Length

Y Variable

Sepal.Width

Cluster count

3

Function	Inserts
`dataTableOutput()`	an interactive table
`htmlOutput()`	raw HTML
`imageOutput()`	image
`plotOutput()`	plot
`tableOutput()`	table
`textOutput()`	text
`uiOutput()`	a Shiny UI element
`verbatimTextOutput()`	text

RStudio

*Output()

To display output, add it to `fluidPage()` with an `*Output()` function

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

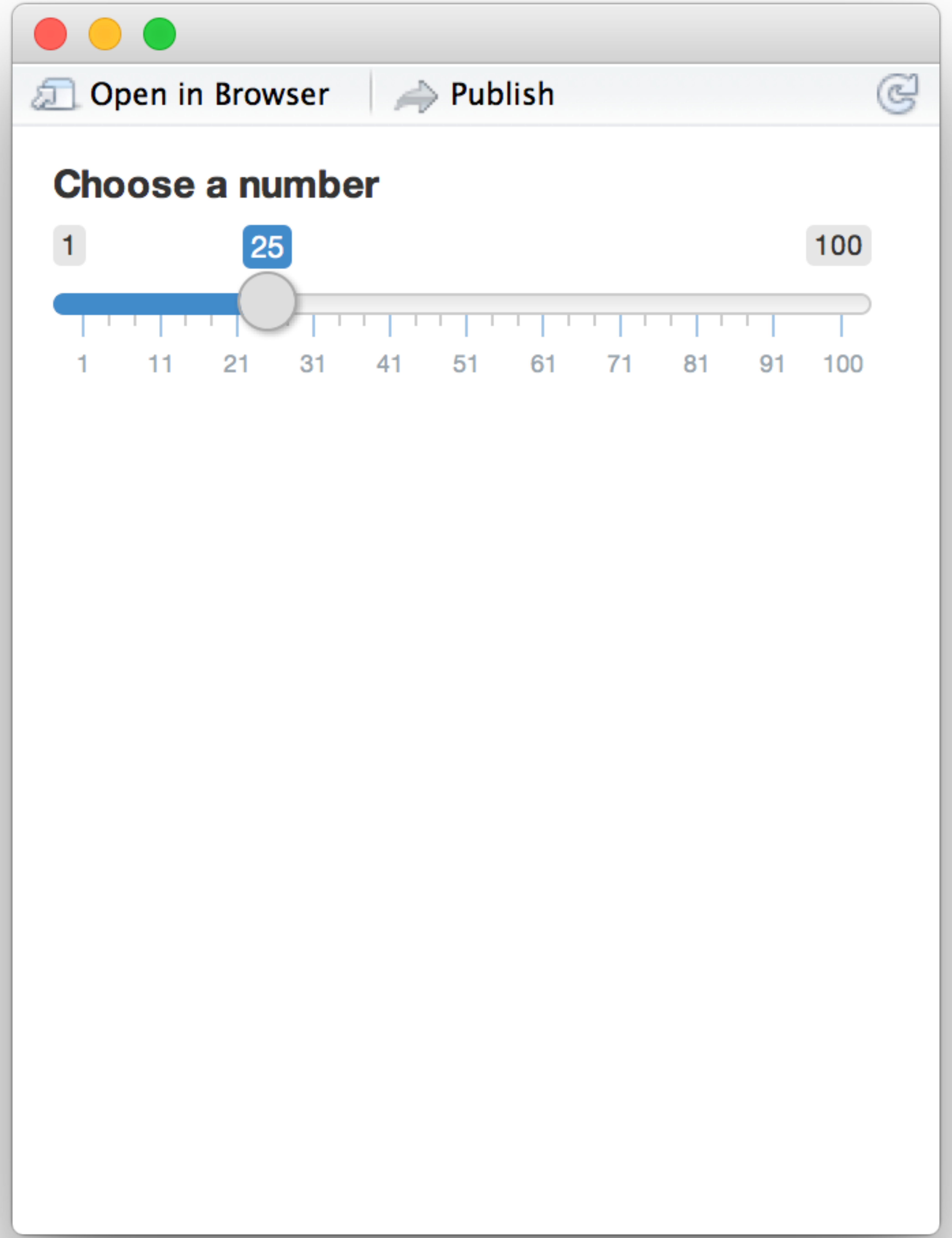
the type of output to display

name to give to the output object

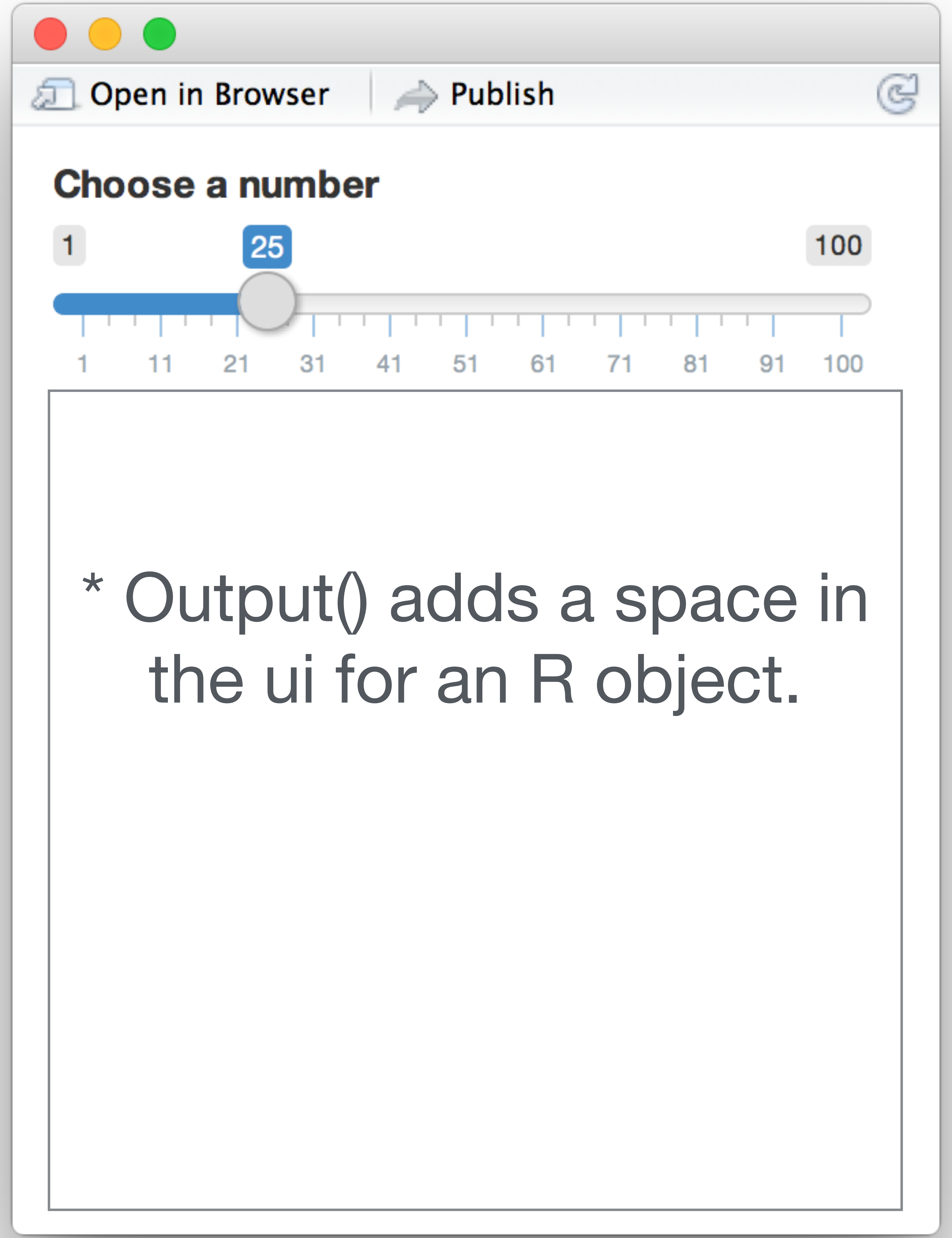
```r
library(shiny)

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)
```

Comma between arguments

```
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)
```

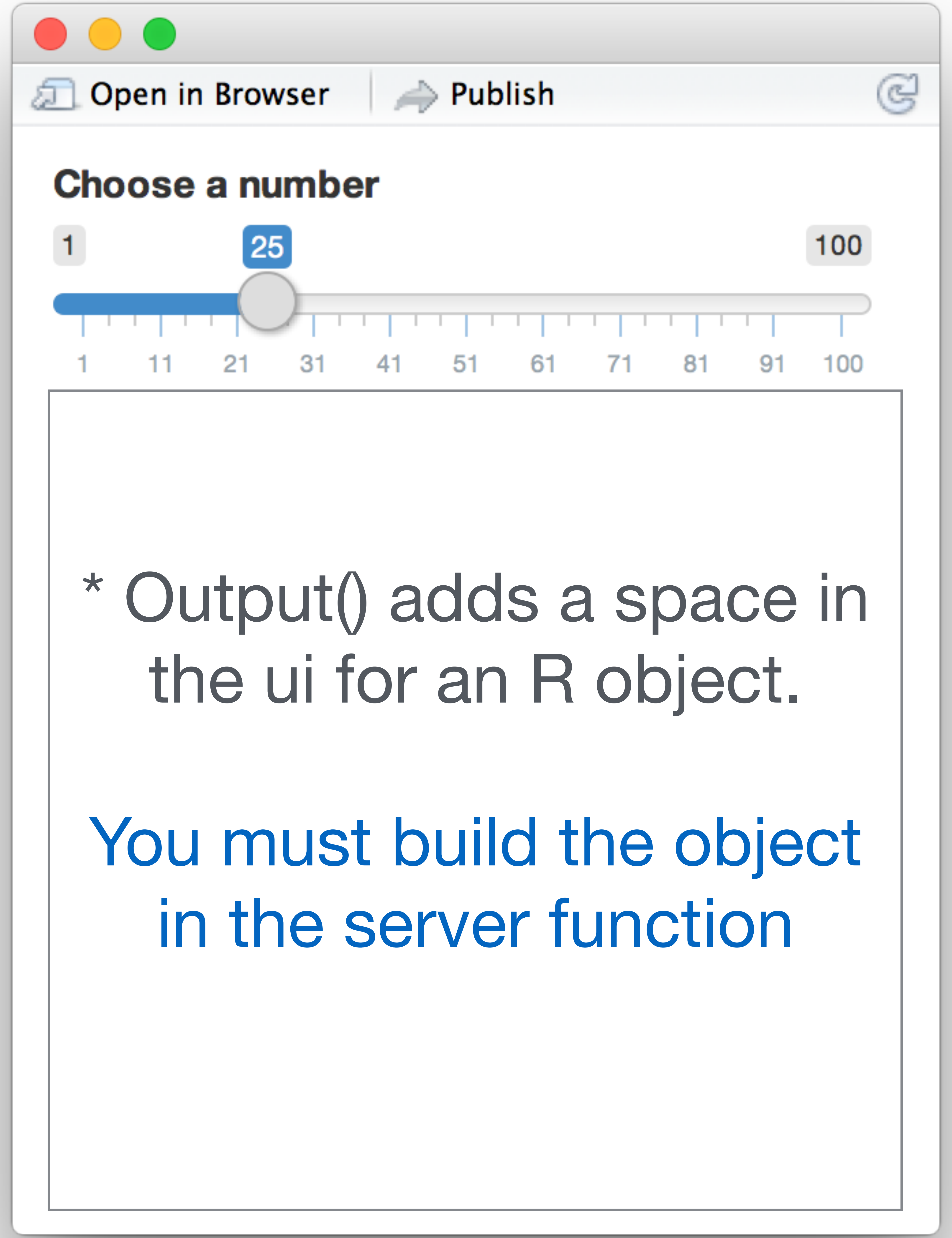


```r
library(shiny)

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)
```



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

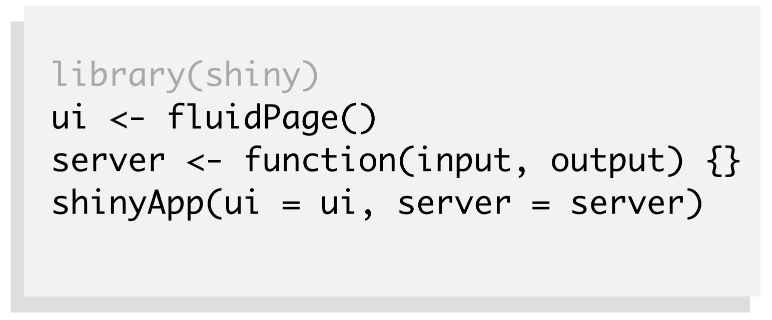
```
library(shiny)

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)
```



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

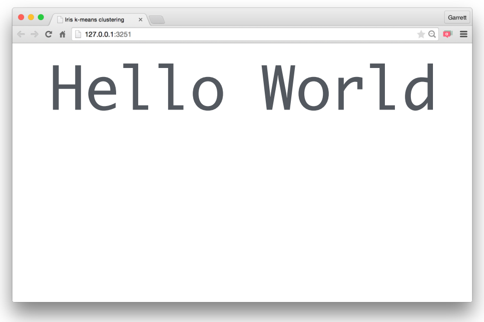
You must build the object in the server function

Recap



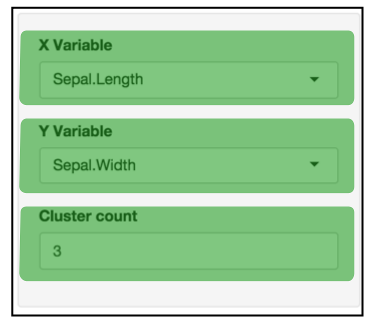
```
library(shiny)
ui <- fluidPage()
server <- function(input, output) {}
shinyApp(ui = ui, server = server)
```

Begin each app with the template

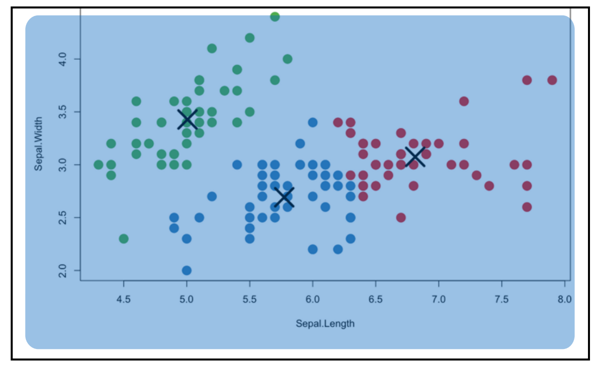


Hello World

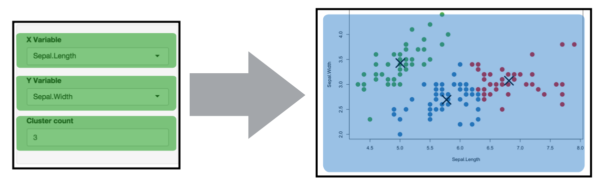
Add elements as arguments to **fluidPage()**



Create reactive inputs with an ***Input()** function



Display reactive results with an ***Output()** function



Assemble outputs from inputs in the server function

Tell the

server

how to assemble inputs into outputs

Use **3 rules** to write the server function

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



}
```

1

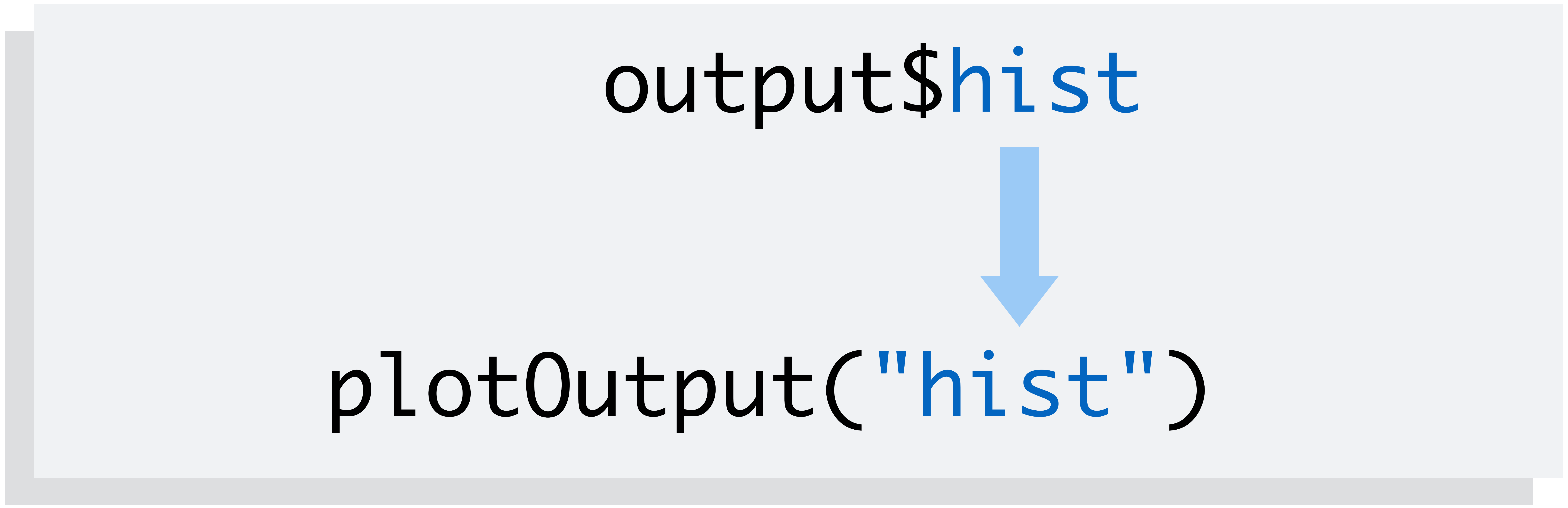
Save objects to display to output$

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


}
```

RStudio

1

Save objects to display to output$



```
output$hist

plotOutput("hist")
```

2

Build objects to display with **render*()**

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

  })
}
```

Use the **render*()** function that creates the type of output you wish to make.

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

render*()

Builds reactive output to display in UI

```
renderPlot({ hist(rnorm(100)) })
```

type of object to build

code block that builds the object

2

Build objects to display with **render*()**

```
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(100))
  })
}
```

2

Build objects to display with **render*()**

```
server <- function(input, output) {
  output$hist <- renderPlot({
    title <- "100 random normal values"
    hist(rnorm(100), main = title)
  })
}
```

RStudio

3

Access **input** values with input$

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

3

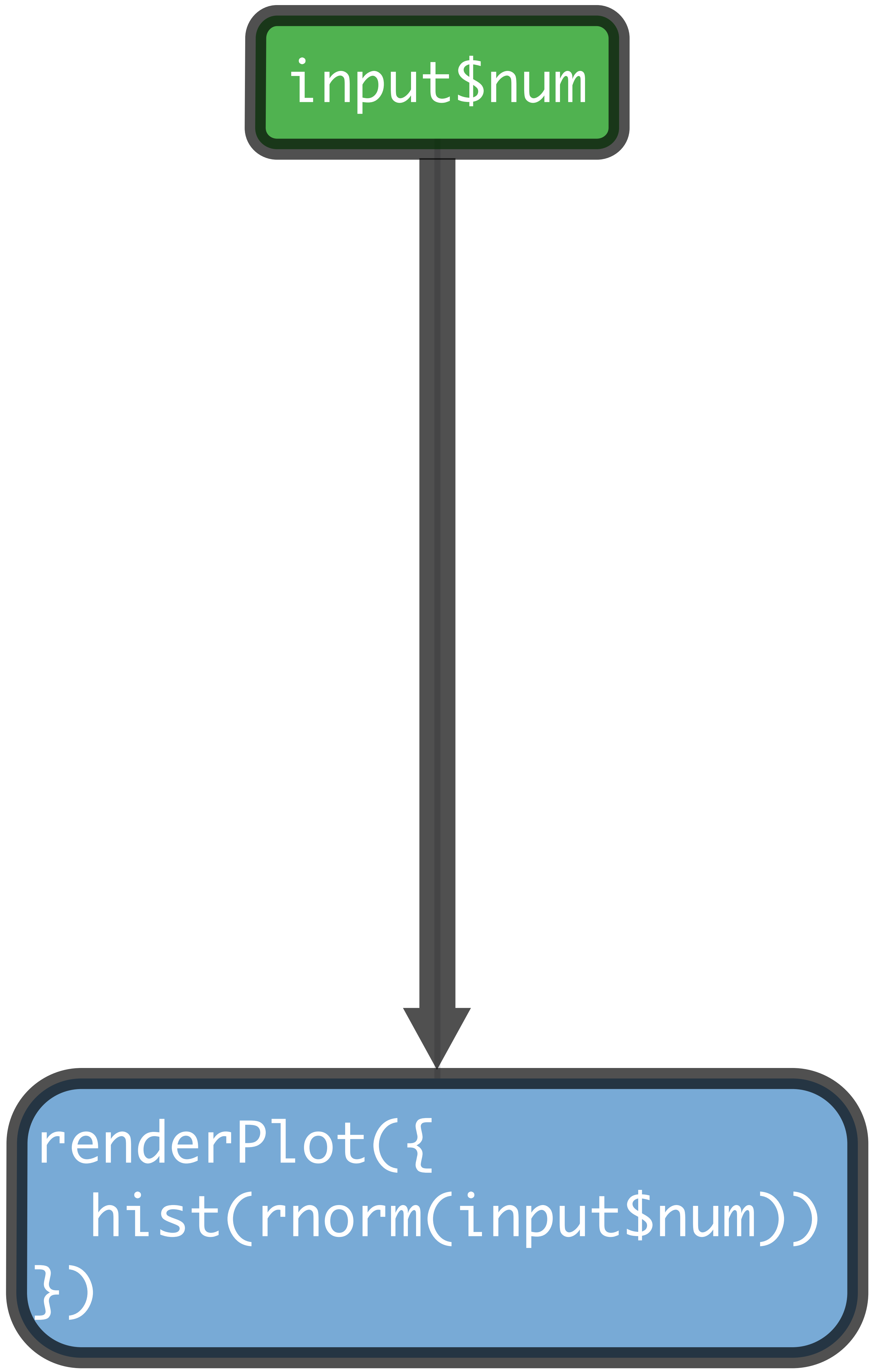
Access **input** values with input$

```
sliderInput(inputId = "num",…)

                          input$num
```

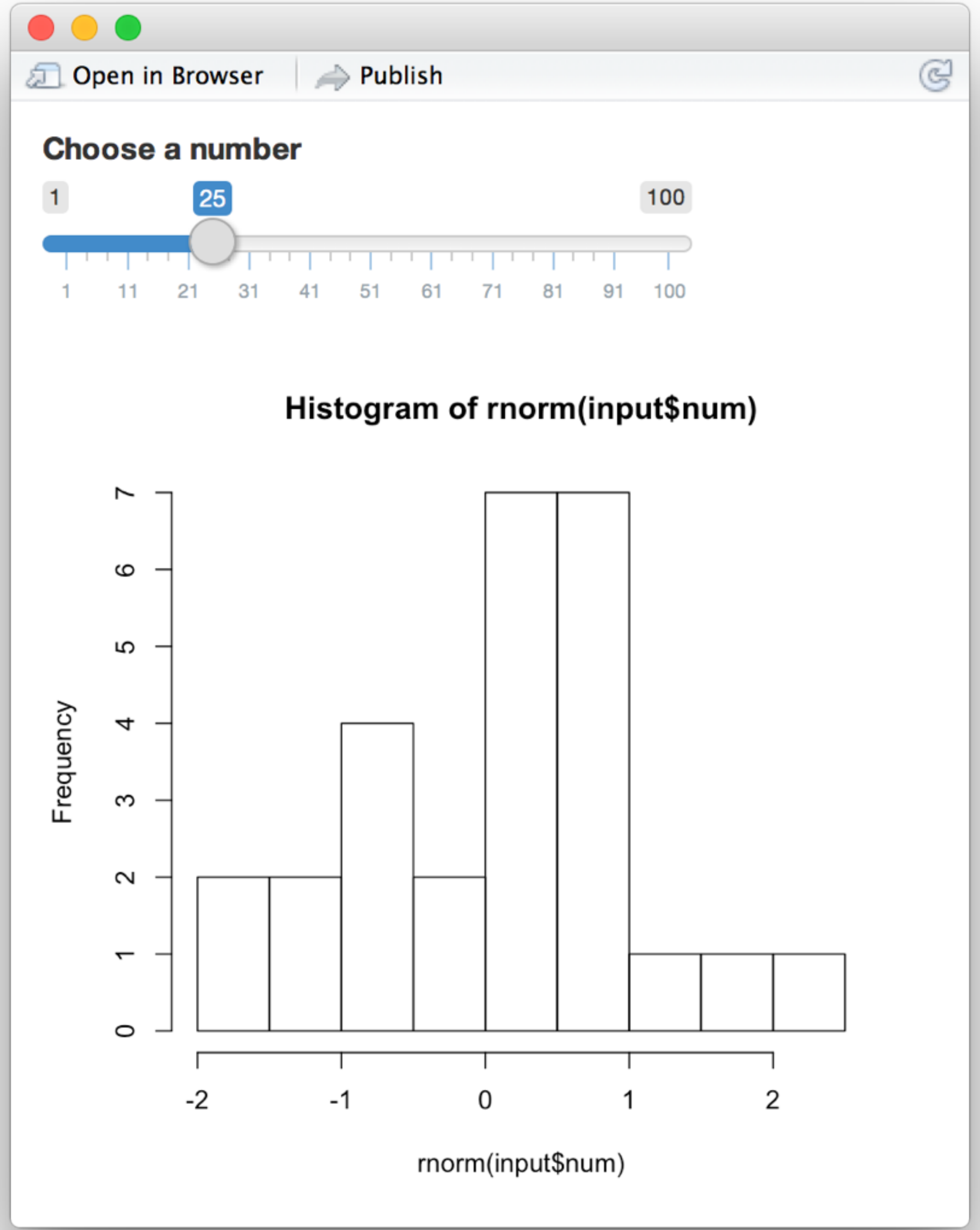
Reactivity 101

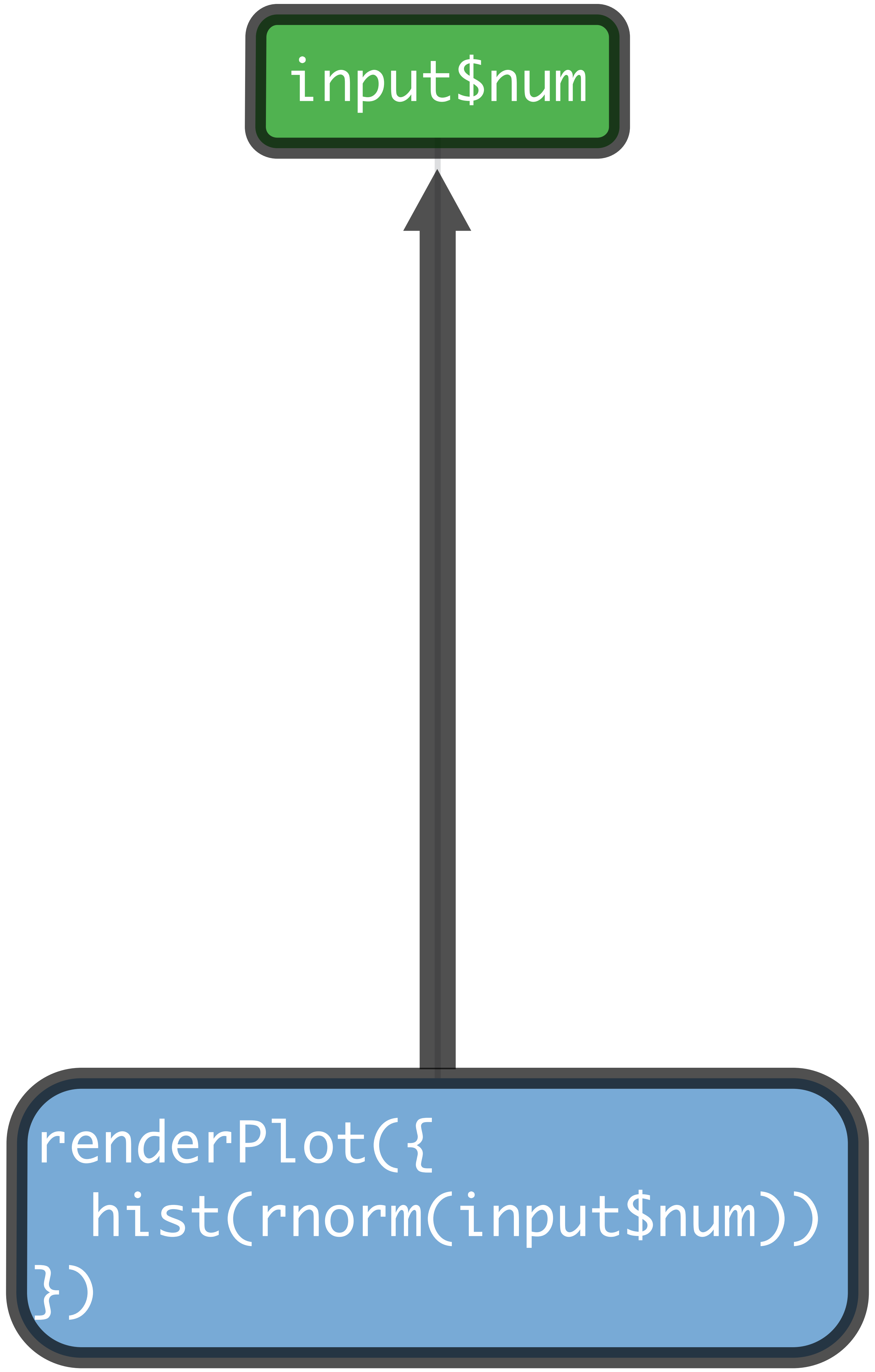
Reactivity automatically occurs whenever you use an input value to render an output object

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



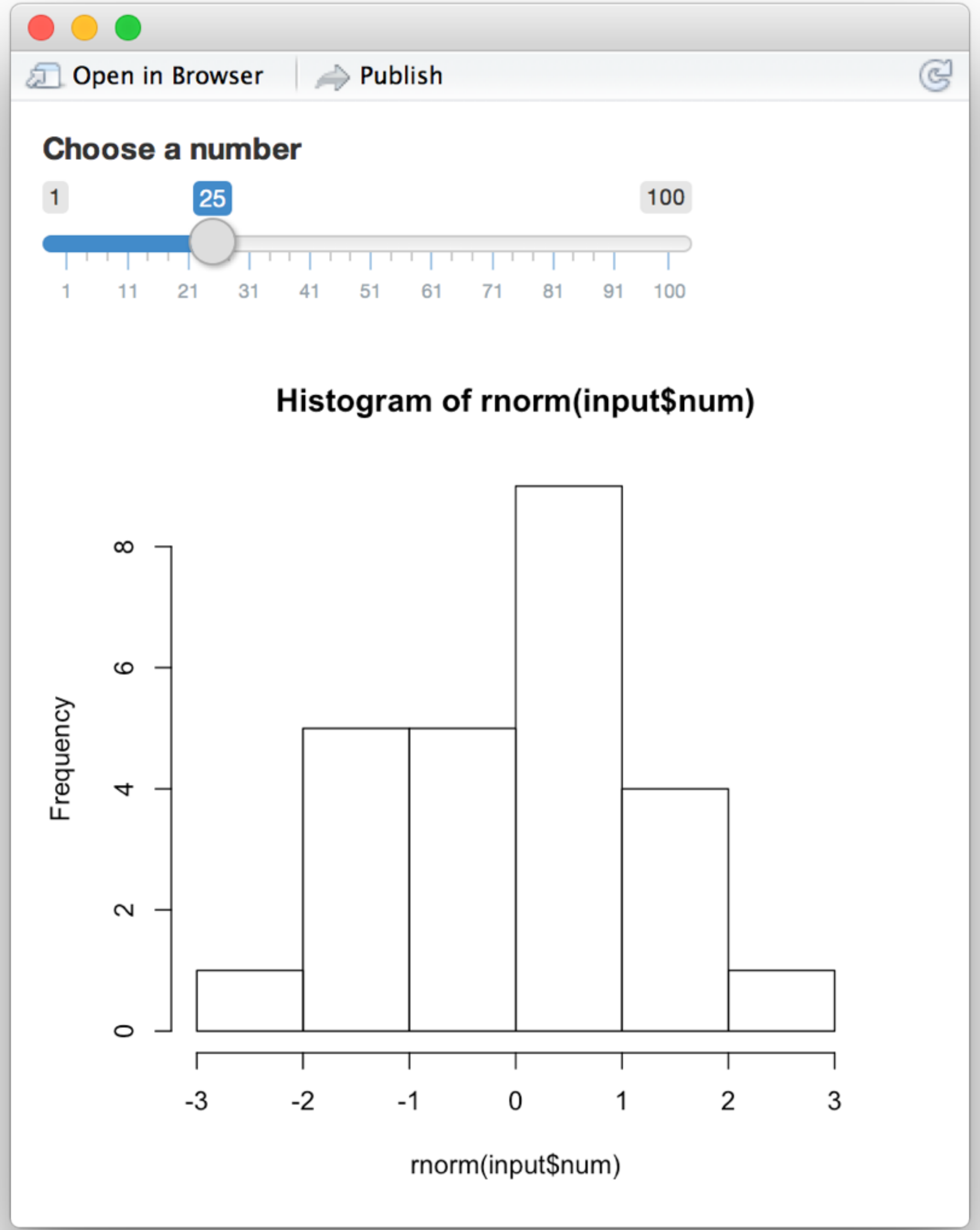
```
renderPlot({
  hist(rnorm(input$num))
})
```





```
input$num
```

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



Recap: Server

Use the server function to assemble inputs into outputs. Follow 3 rules:

`output$hist <-`

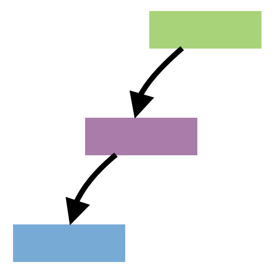
1. Save the output that you build to **output$**

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

2. Build the output with a **render*()** function

`input$num`

3. Access input values with **input$**



Create reactivity by using **Inputs** to build **rendered Outputs**