Building Shiny Apps

W. Evan Johnson, Ph.D.
Professor, Division of Infectious Disease
Director, Center for Data Science
Rutgers University – New Jersey Medical School

2024-05-23

What is Shiny?

`Shiny` is an R package that allows you to easily create rich, interactive web apps around your R functions and packages. `Shiny` allows you to take your work in R and disseminate it via a web browser so that anyone can use it. `Shiny` makes you look awesome by making it easy to produce polished web apps with a minimum amount of pain.

(Credit: Some of the images and slide text were taken and adapted from https://mastering-shiny.org and RStudio's `Shiny` tutorials)

What is Shiny?

`Shiny` is designed primarily with data scientists in mind, and to that end, you can create pretty complicated `Shiny` apps with no knowledge of HTML, CSS, or JavaScript. On the other hand, `Shiny` doesn't limit you to creating trivial or prefabricated apps: its user interface components can be easily customized or extended, and its server uses reactive programming to let you create any type of back end logic you want.

What is Shiny?

Some of the things people use `Shiny` for are:

- Create dashboards
- Replace hundreds of pages of PDFs with interactive apps
- Communicate complex models informative visualizations
- Provide self-service data analysis for common workflows
- Create interactive demos for teaching statistics and data science

In short, `Shiny` gives you the ability to pass on some of your `R` superpowers to anyone who can use the web.

Getting Started

To create your first `Shiny` App you wil need:

- Install R: Come on now!!
- Install RStudio (strongly recommended): https://www.rstudio.com/products/rstudio/download
- The R `Shiny` package: `install.packages("shiny")`
- Knowledge and experience in developing R packages (recommended)
- Git and GitHub (recommended): to share your Apps

Getting Started

If you haven't already installed `Shiny`, install it now with:

```
install.packages("shiny")
```

If you've already installed `Shiny`, use the following to check that you have version 1.5.0 or greater:

```
packageVersion("shiny")
```

```
## [1] '1.8.1.1'
```

Shiny Introduction

RStudio's Shiny Showcase is an exciting assembly of exciting apps!

Or we can start with a very simple example

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

Shiny Introduction

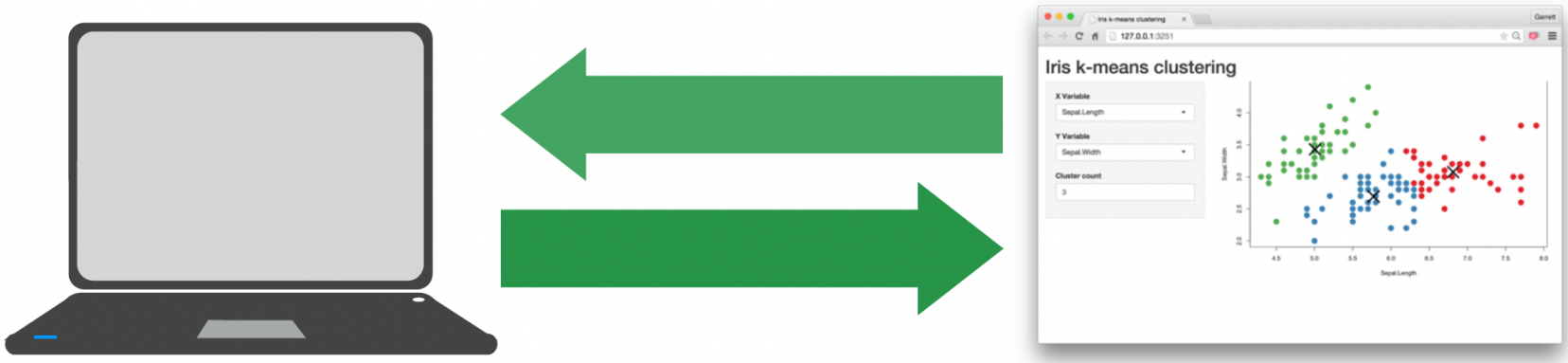
Or an example from Dr. Johnson's research:

```
devtools::install_github("compbiomed/animalcules")
library(animalcules)
run_animalcules()
```

(Warning: this might take a while to load package dependencies!)

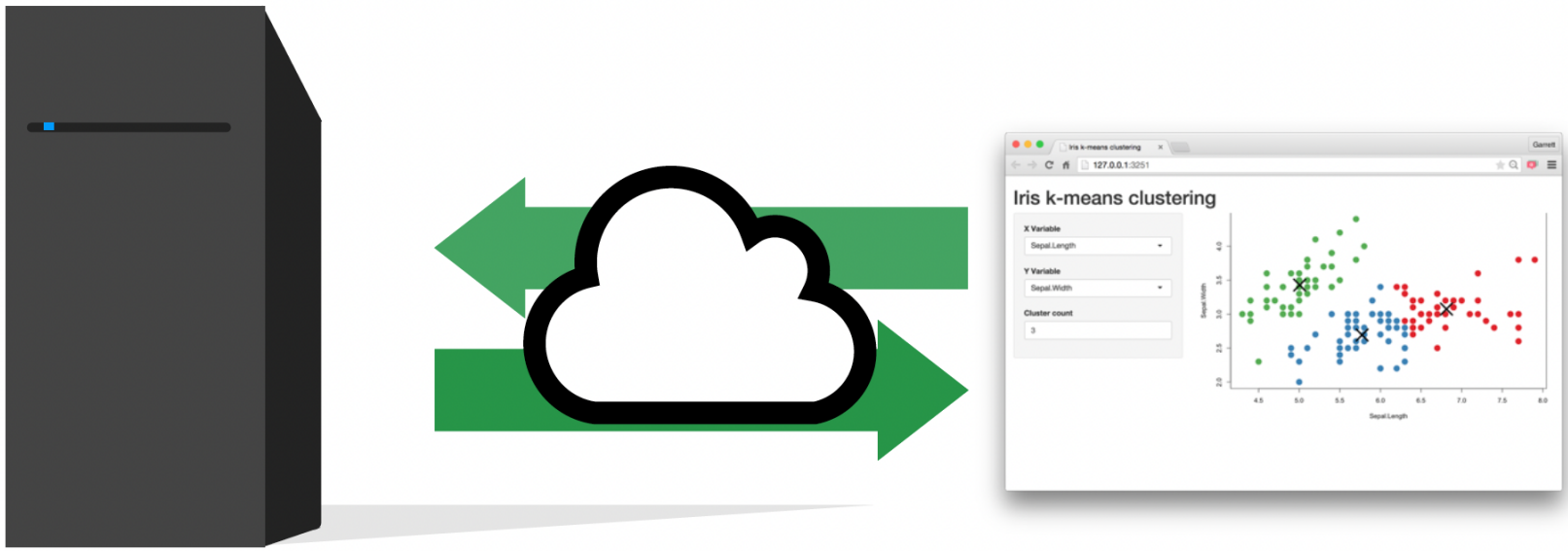
Shiny Architecture

Every Shiny app is maintained by a computer running R



Shiny Architecture

Every Shiny app is maintained by a computer running R



Shiny Architecture

Introduction to shiny development

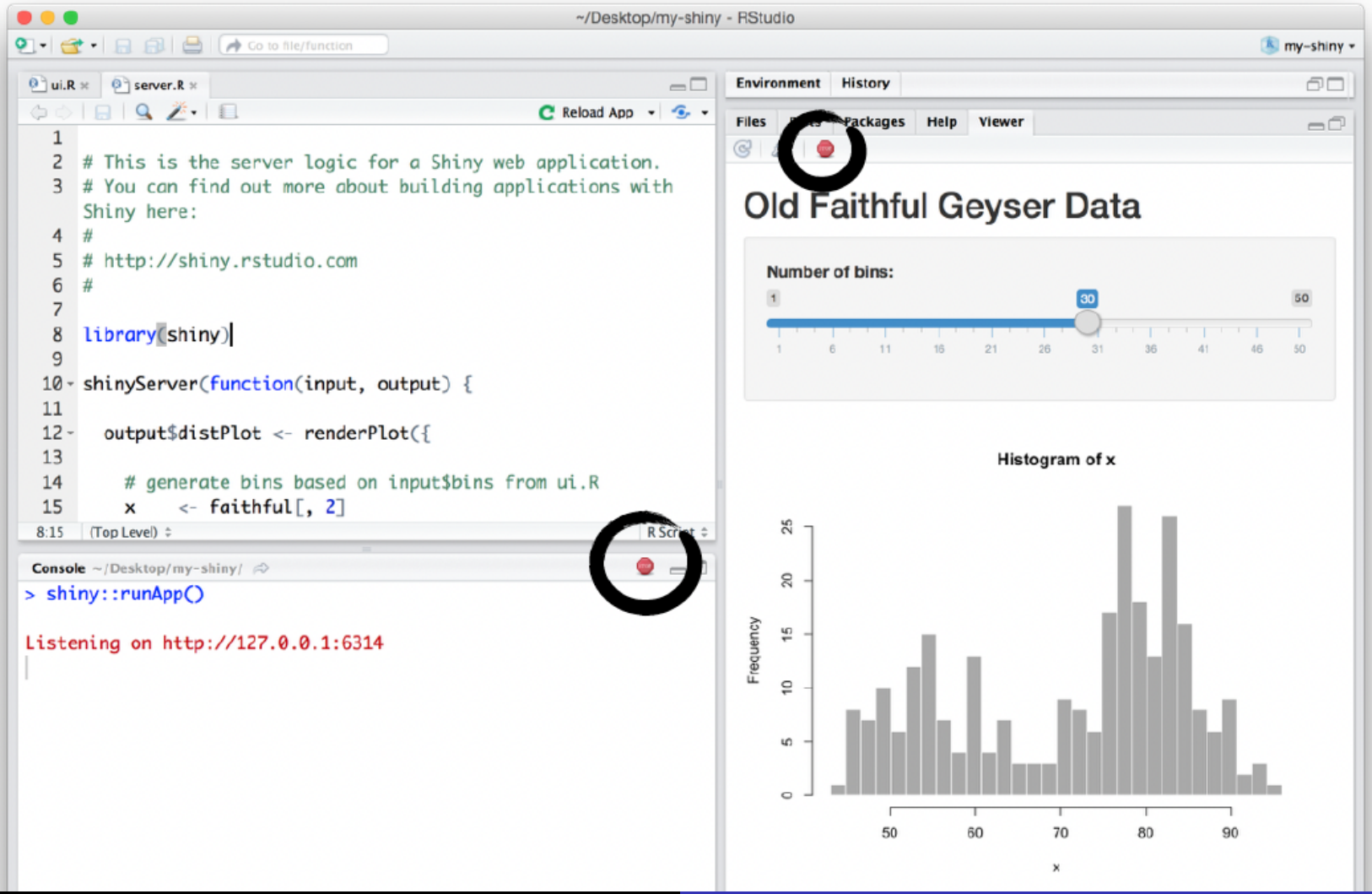
Here we'll create a simple `Shiny` app, staring with the minimum boilerplate, followed by the two key components of every `Shiny` app: the UI (short for user interface) which defines how your app looks, and the server function which defines how your app works. `Shiny` uses **reactive** programming to automatically update outputs when inputs change.

Template: shortest viable app

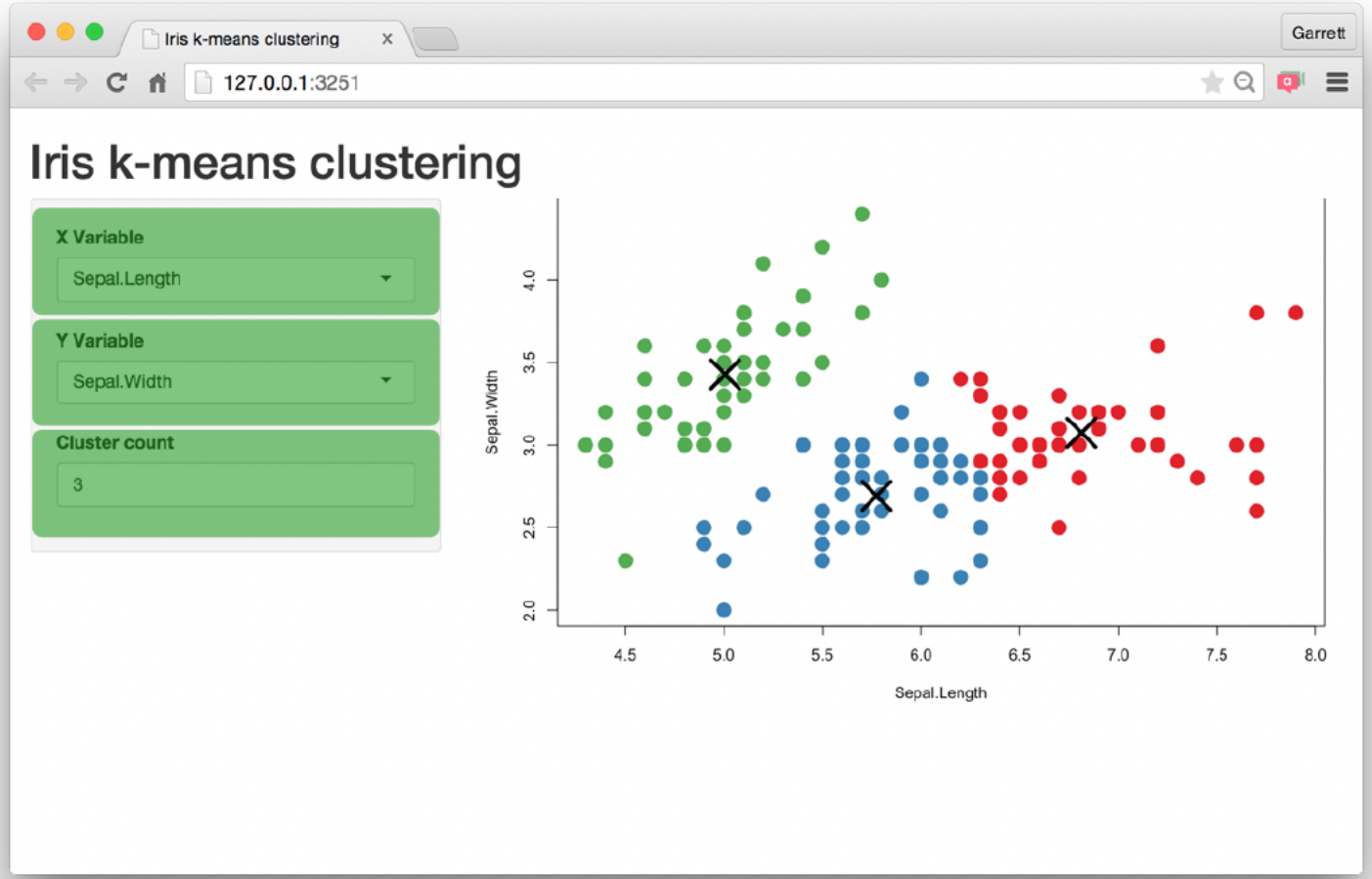
The simplest way to start an app is to create a new directory, and add a single file called app.R, with the following code:

```
library(shiny)

ui <- fluidPage()

server <- function(input, output) {}

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

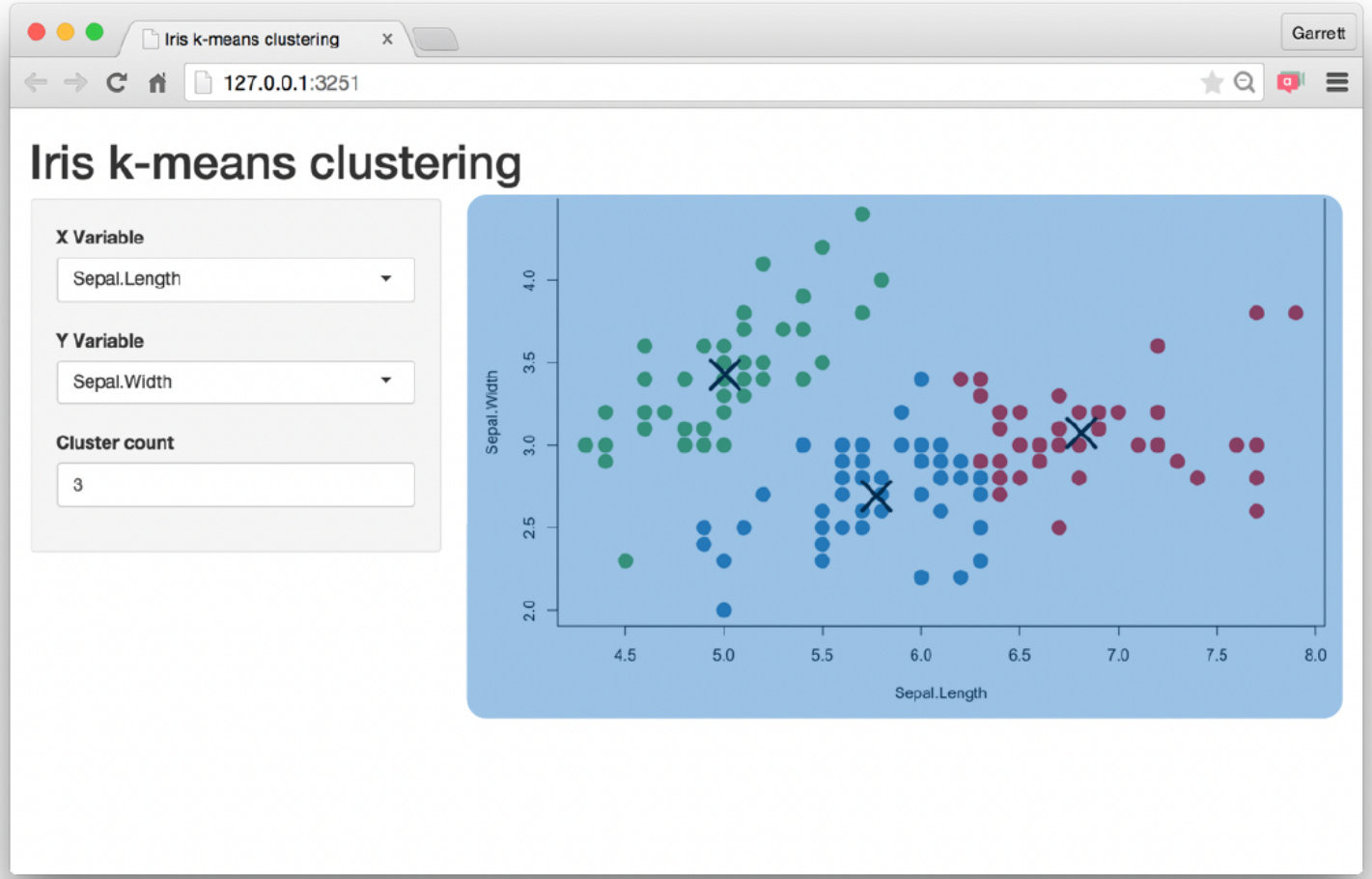
Close an app


~/Desktop/my-shiny - RStudio
ui.R
server.R
Reload App
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]
Console ~/Desktop/my-shiny/
> shiny::runApp()
Listening on http://127.0.0.1:6314
Environment
History
Files
Packages
Help
Viewer
Old Faithful Geyser Data
Number of bins:
Histogram of x
Frequency
x

Build your app around inputs and outputs



Build your app around inputs and outputs



Adding elements to the UI

Add elements to your app as arguments to fluidPage()

```
library(shiny)

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

server <- function(input, output) {}

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

Adding elements to the UI

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

```
library(shiny)

ui <- fluidPage(
  "Hello World!"
  )

server <- function(input, output) {}

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

Input functions

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

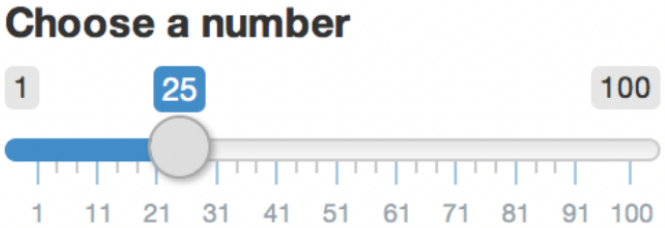
```
sliderInput(inputId = "num",
            label = "Choose a number",
            value = 25, min = 1, max = 100)
```

Here is the actual HTML code for this slider:

```
<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="tru
  data-keyboard-step="1.01010101010101"/>
</div>
```

Shiny buttons

Syntax




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

- `inputId`: input name (for internal use)
- Notice: Id not ID
- `label`: label to display
- `…`: input specific arguments

`?sliderInput`

Input functions in an app

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)
```

Shiny buttons

Buttons

Action

Submit

`actionButton()`
`submitButton()`

Single checkbox

Choice A

`checkboxInput()`

Checkbox group

Choice 1
Choice 2
Choice 3

`checkboxGroupInput()`

Date input

2014-01-01

`dateInput()`

Date range

2014-01-24 to 2014-01-24

`dateRangeInput()`

File input

Choose File No file chosen

`fileInput()`

Numeric input

1

`numericInput()`

Password Input

`passwordInput()`

Radio buttons

Choice 1
Choice 2
Choice 3

`radioButtons()`

Select box

Choice 1

`selectInput()`

Sliders

0 50 100

0 25 75 100

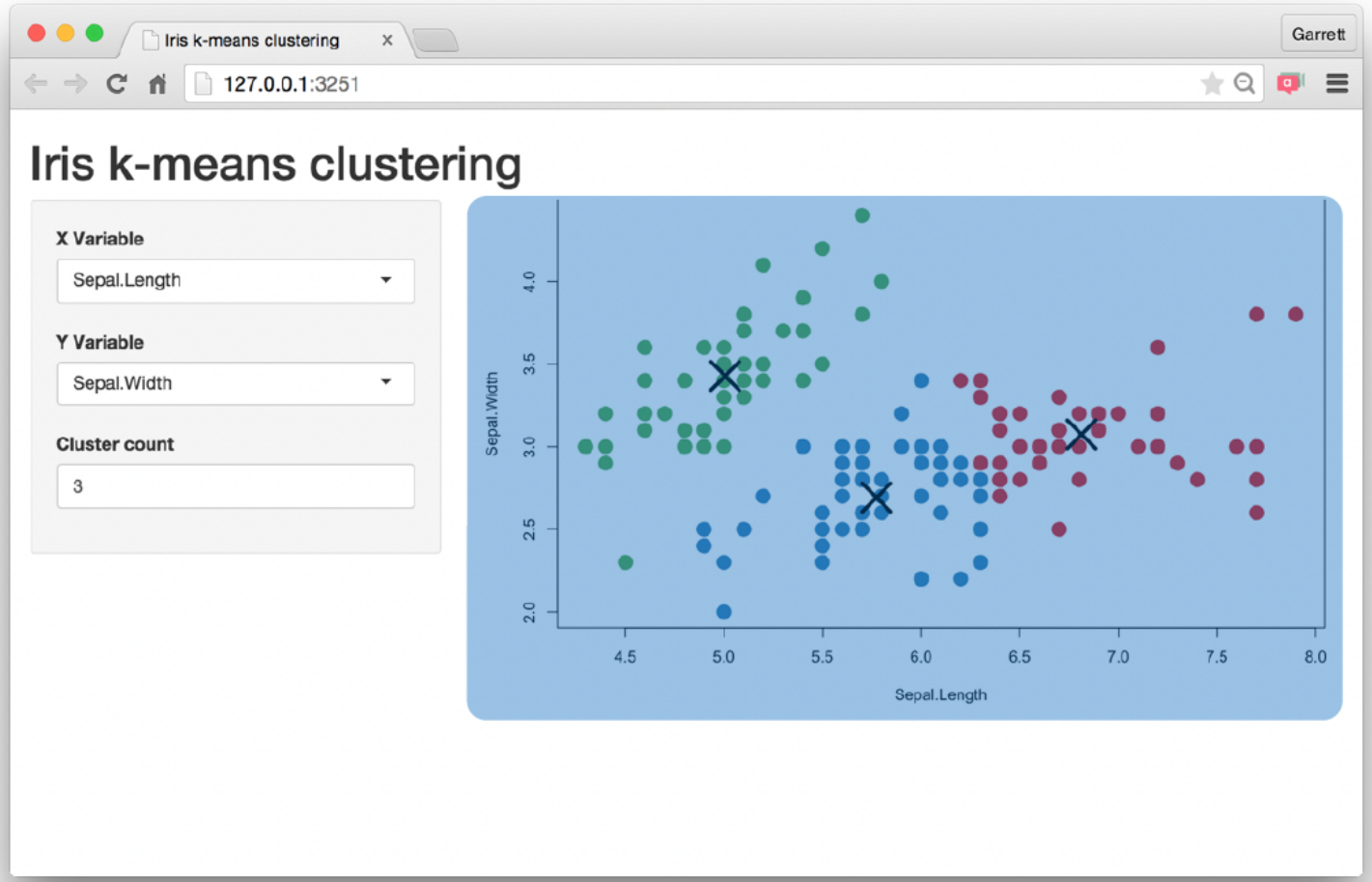
`sliderInput()`

Text input

Enter text...

`textInput()`

Build your app around inputs and outputs



Shiny outputs

Shiny outputs

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

Adding outputs

Create an output with an **Output()** function.

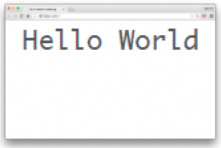
```
library(shiny)

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

server <- function(input, output) {}

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

Note that you must build the output in the server first!

Recap

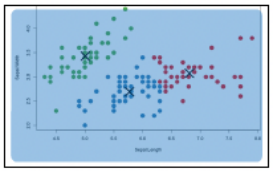
```
library(shiny)
ui <- fluidPage()
server <- function(input, output) {}
shinyApp(ui = ui, server = server)
```

Begin each app with the template

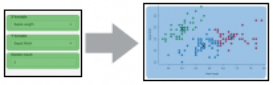


Add elements as arguments to **fluidPage()**

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



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



Assemble outputs from inputs in the server function

Building the server

There are 3 basic rules for your **server** function:

1. Access input values with **input$**
2. Save objects to display as **output$**
3. Build objects to display with **render()**

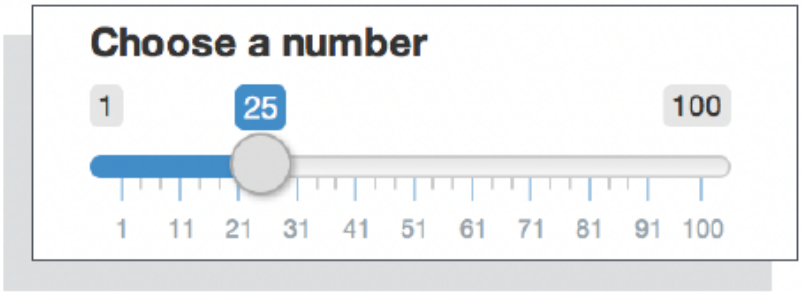
Server inputs

You can access inputs from the UI using **input$**

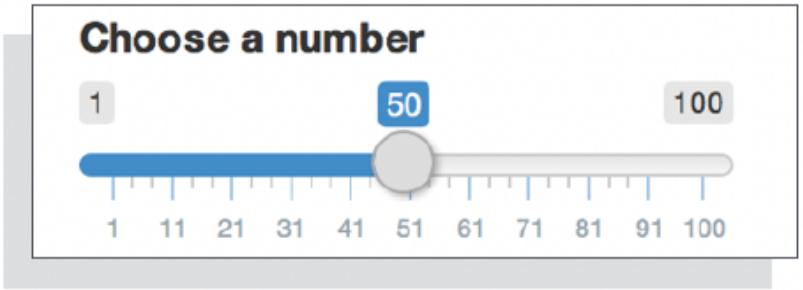
```
sliderInput(inputId = "num",...)
```

```
input$num
```

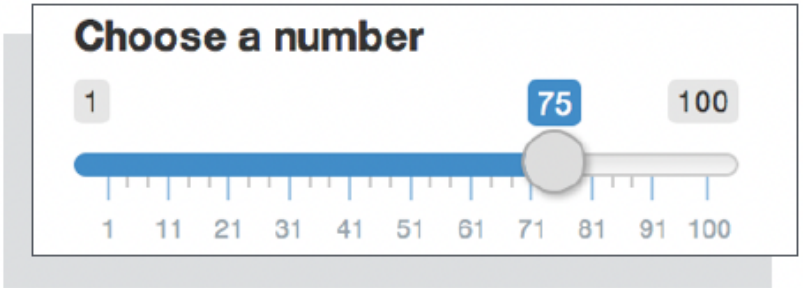
The input value changes whenever a user changes the input.



```
input$num = 25
```



```
input$num = 50
```



```
input$num = 75
```

Server outputs You can save your outputs to return to the UI

Server outputs

You can save your outputs to return to the UI using **output$**

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


}
```

Rendering server outputs

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

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

  })
}
```

Rendering server outputs

render*()

Builds reactive output to display in UI

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

type of object to build

code block that builds the object

Rendering server outputs

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

Reactivity 101

The term **reactivity** refers to the idea that the inputs and outputs are connected to each other: when you change and input, you change all outputs that require the input.

Reactivity is an exciting charateristic, but can also get you in trouble!

Recap: Server

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

`output$hist <-`

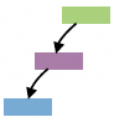
1. Save the output that you build to **output$**

```
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**

Complete Shiny app

Now putting everything together:

```
library(shiny)

ui <- fluidPage(
  sliderInput(inputId = "num",
    label = "Choose a number",
    value = 25, min = 0, max = 100),
  plotOutput("hist")
  )

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

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

Multi-file apps:

Now lets get more organized, create a **ui.R** file

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

And a **server.R** file:

```
#server.R
library(shiny)
server <- function(input, output){
  output$hist <- renderPlot({
    hist(rnorm(input$num))
    })
  }
```

Multi-file apps:

And then finally the program file, **app.R**:

```
#app.R
library(shiny)

source("ui.R")
source("server.R")

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

Building apps in R packages:

In an R package, create an `inst/shiny` directory and place the `Shiny` code there. Also:

- Write the package functions first!
- This includes all analytics AND plotting functions
- `Shiny` app should call functions, provide inputs, display results
- Operate on high-quality R data objects!

Also, see examples from Dr. Johnson's research, e.g.:
https://github.com/compbiomed/animalcules

Session Info

```
## R version 4.4.0 (2024-04-24)
## Platform: aarch64-apple-darwin20
## Running under: macOS Sonoma 14.2.1
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib;  LAPACK vers
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## time zone: America/Denver
## tzcode source: internal
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## loaded via a namespace (and not attached):
##  [1] compiler_4.4.0    fastmap_1.1.1     cli_3.6.2         tools_4.4.0
##  [5] htmltools_0.5.8.1 rstudioapi_0.16.0 yaml_2.3.8        rmarkdown_2.26
##  [9] knitr_1.46        xfun_0.43         digest_0.6.35     rlang_1.1.3
## [13] evaluate_0.23
```