

# R Shiny Part II

## Statistical Computing & Programming

### Shawn Santo

# Supplementary materials

Full video lecture available in Zoom Cloud Recordings

Additional resources

- Shiny reactivity
- Shiny code quality

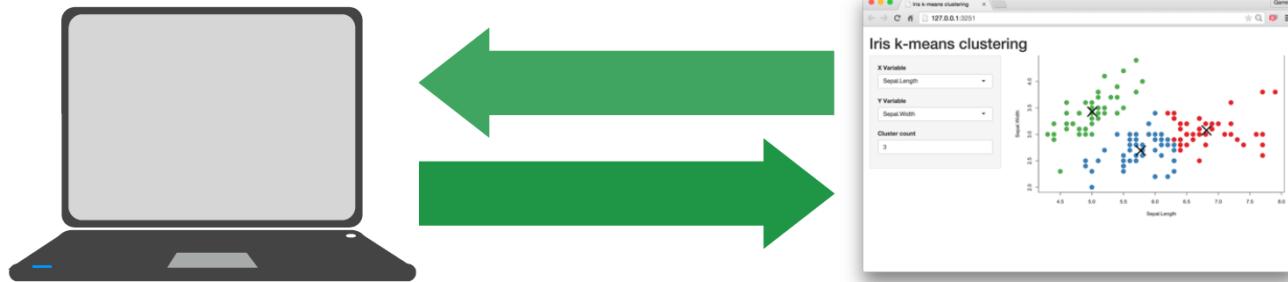
# Recall

# What is Shiny?

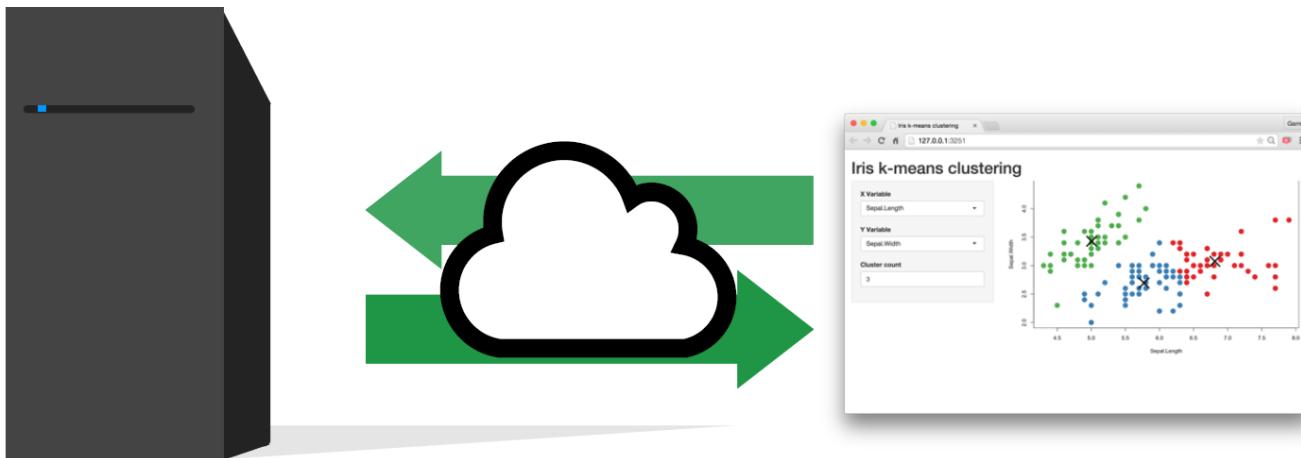
- Shiny is an R package.
- Build web-based apps with R in RStudio.
- Shiny can incorporate CSS themes and JavaScript actions.



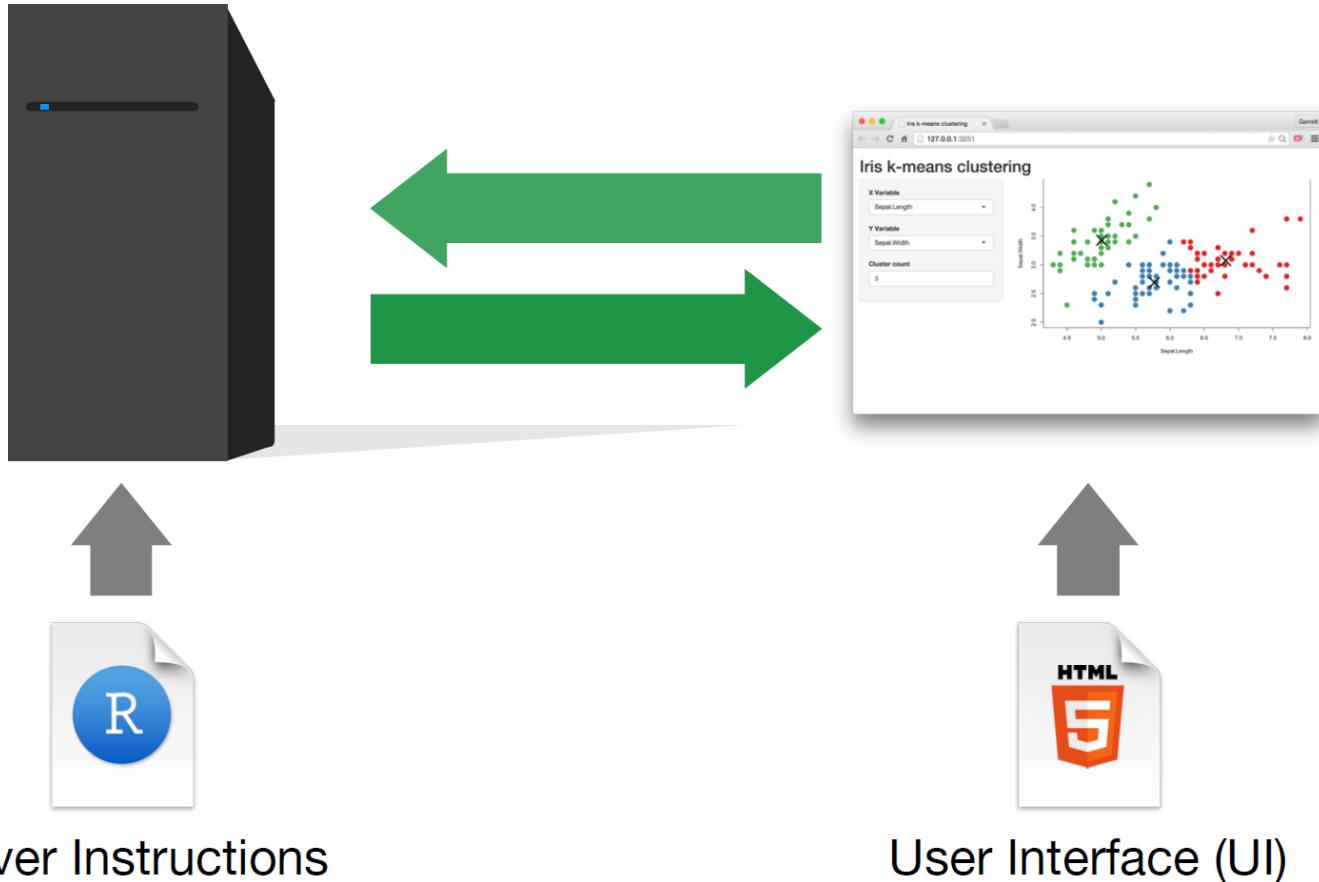
# How does Shiny work?



# How does Shiny work?



# How does Shiny work?



Server Instructions

User Interface (UI)

# Main components of Rshiny

```
# Load package shiny
library(shiny)
# Define UI for application
ui <- fluidPage(
  )

# Define server logic
server <- function(input, output)

}

# Build and run the application
shinyApp(ui = ui, server = server)
```

- Function `fluidPage()` creates a dynamic HTML user interface you see when you look at an RShiny app. Convention is to save this as an object named `ui`.
- Function `server()` is user-defined and contains R commands your computer or external server need to run the app.
- Function `shinyApp()` builds the app based on the user interface and server pair of code.

# User interface review

- Build the user interface inside function `fluidPage()` and save it as an object named `ui`.
- Function `fluidPage()` scales its components in real-time to fill all available browser width - dynamic HTML user interface.
- Build inputs with `*Input(inputId, label, ...)`.
- Build outputs with `*Output(outputId, ...)`.
- Separate multiple inputs and outputs with commas.
- Run your app after each added input or output to minimize complications later on.

# Function server() review

- The server function does the work in terms of building and rebuilding R objects.
- Save output you build to `output$<outputId>`.
- Build output with a `render*()` function.
- Access inputs with `input$<inputId>`.
- Multiple outputs can be placed in the server function.
- Reactivity happens automatically when you use inputs to build rendered outputs.

# UI Layouts

# Layouts

- Use layout functions to position elements in your app.
- Use panels to group elements into a single unit for aesthetic or functional purposes.
- Design your own layout or use a packaged layout.

# Rows with fluidRow()

```
ui <- fluidPage(  
  fluidRow(  
    # add inputs/outputs to row 1  
  ),  
  fluidRow(  
    # add inputs/outputs to row 2  
  ),  
  fluidRow(  
    # add inputs/outputs to row 3  
  )  
)
```

# Rows and columns

```
ui <- fluidPage(  
  
  fluidRow(  
    column(width = 5,  
           # add inputs/outputs  
           # column width 5  
    ),  
    column(width = 7,  
           # add inputs/outputs  
           # column width 7  
    )  
  ),  
  
  fluidRow(  
    column(width = 8, offset = 2,  
           # add inputs/outputs  
           # 2 units in from left  
    )  
  )  
)
```

# Rows and columns example

```
ui <- fluidPage(  
  
  fluidRow(  
    column(width = 5,  
           passwordInput(inputId = "pass",  
                           label = "Enter password:"),  
           actionButton(inputId = "passbtn",  
                           label = "Submit password")  
    ),  
  
    column(width = 7,  
           paste("Add some description in row 1",  
                  "of the column with a width of 7",  
                  "here...", sep = " "))  
  ),  
  
  fluidRow(  
    column(width = 8, offset = 2,  
           checkboxGroupInput(inputId = "checks",  
                           label = "",  
                           choices = c("Choice 1",  
                                      "Choice 2",  
                                      "Choice 3",  
                                      "Choice 4")),  
           "Add some more text in Row 2 here..."  
    )  
  )  
)
```

# Panels

## **absolutePanel()**

Panel position set rigidly (absolutely), not fluidly

## **conditionalPanel()**

A JavaScript expression determines whether panel is visible or not.

## **fixedPanel()**

Panel is fixed to browser window and does not scroll with the page

## **headerPanel()**

Panel for the app's title, used with pageWithSidebar()

## **inputPanel()**

Panel with grey background, suitable for grouping inputs

## **mainPanel()**

Panel for displaying output, used with pageWithSidebar()

## **navlistPanel()**

Panel for displaying multiple stacked tabPanels(). Uses sidebar navigation

## **sidebarPanel()**

Panel for displaying a sidebar of inputs, used with pageWithSidebar()

## **tabPanel()**

Stackable panel. Used with navlistPanel() and tabsetPanel()

## **tabsetPanel()**

Panel for displaying multiple stacked tabPanels(). Uses tab navigation

## **titlePanel()**

Panel for the app's title, used with pageWithSidebar()

## **wellPanel()**

Panel with grey background.

# Packaged layout: sidebarLayout()

```
ui <- fluidPage(  
  
  # give a title in quotes  
  titlePanel(),  
  
  sidebarLayout(  
    sidebarPanel(  
  
      # inputs/outputs  
    ),  
  
    mainPanel(  
      # inputs/outputs  
    )  
  )  
)
```

Sidebar Layout

Sidebar  
Panel

Main Panel

# Packaged layout: navbarPage()

```
ui <- fluidPage(  
  
  navbarPage(title = "Nav Bar Layout",  
  
    tabPanel("Tab 1",  
            # add inputs/outputs  
            ),  
  
    tabPanel("Tab 2",  
            # add inputs/outputs  
            ),  
  
    tabPanel("Tab 3",  
            # add inputs/outputs  
            ),  
  
    tabPanel("Tab 4",  
            # add inputs/outputs  
            ))  
  )
```



Add inputs and outputs to Tab 1

# Packaged layout: navbarPage()

```
ui <- fluidPage(  
  
  navbarPage(title = "Nav Bar Layout",  
  
    tabPanel("Tab 1",  
            # add inputs/outputs  
            ),  
  
    tabPanel("Tab 2",  
            # add inputs/outputs  
            ),  
  
    tabPanel("Tab 3",  
            # add inputs/outputs  
            ),  
  
    tabPanel("Tab 4",  
            # add inputs/outputs  
            ))  
  )
```



Add inputs and outputs to Tab 2

# Layout recap

- Use `fluidRow()` to arrange elements in rows; use `columns()` to arrange elements in columns, where total width is 12
- Use `sidebarPanel()` and `mainPanel()` to partition app with the packaged layout function `sidebarLayout()`
- Use `tabPanel()` with packaged layouts `navbarPage()` or `navbarMenu()`.

# Reactivity

# Frequency of code execution

- Shiny will run the whole script the first time your app is launched.
- Each time a new user visits your app, Shiny runs the server function again, one time.
- As users interact with widgets, Shiny will re-run the corresponding R expressions that depend on a widget whose value was changed.

# Consider the simple app

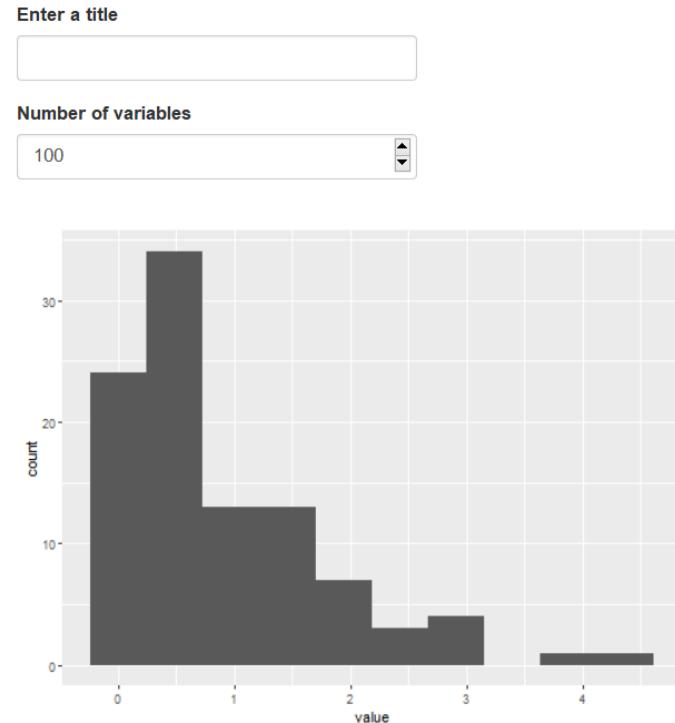
```
library(shiny)
library(tidyverse)

# Build UI
ui <- fluidPage(
 textInput(inputId = "title",
            label = "Enter a title"),
  numericInput(inputId = "num",
               label = "Number of variables",
               value = 100,
               min = 1),
  plotOutput(outputId = "hist")
)

# Define server function
server <- function(input, output) {

  output$hist <- renderPlot({
    ggplot(as_tibble(rexp(input$num)),
           aes(x = value)) +
      geom_histogram(bins = 10) +
      labs(title = input$title)
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```



# Is there a problem?

```
library(shiny)
library(tidyverse)

# Build UI
ui <- fluidPage(
 textInput(inputId = "title",
            label = "Enter a title"),
  numericInput(inputId = "num",
               label = "Number of variables",
               value = 100,
               min = 1),
  plotOutput(outputId = "hist")
)

# Define server function
server <- function(input, output) {

  output$hist <- renderPlot({
    ggplot(as_tibble(rexp(input$num)),
           aes(x = value)) +
      geom_histogram(bins = 10) +
      labs(title = input$title)
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

- Every time you change the title, `labs(title = input$title)` new random numbers will be generated. If a single input changes in a block of code inside a render function, then the entire block of code is re-run.
- This is very inefficient and can cause problems.

# Attempted solution

```
library(shiny)
library(tidyverse)

# Build UI
ui <- fluidPage(
 textInput(inputId = "title",
            label = "Enter a title"),
  numericInput(inputId = "num",
               label = "Number of variables",
               value = 100,
               min = 1),
  plotOutput(outputId = "hist")
)

# Define server function
server <- function(input, output) {

  data <- rexp(input$num)

  output$hist <- renderPlot({
    ggplot(as_tibble(data),
           aes(x = value)) +
      geom_histogram(bins = 10) +
      labs(title = input$title)
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

Error **in** .getReactiveEnvironment()  
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.)

- Reactive inputs must be in a reactive-type function.

# Reactive expressions

The render functions are reactive-type functions. Function `reactive()` builds a reactive object. The object will respond to every reactive source in the code.

Rather than

```
data <- rexp(input$num)
```

use

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

A reactive expression has two special properties:

1. call a reactive expression like a function, `data()`;
2. reactive expressions cache their values, the value is retained until it becomes invalidated.

# Solution

```
library(shiny)
library(tidyverse)

# Build UI
ui <- fluidPage(
 textInput(inputId = "title",
            label = "Enter a title"),
  numericInput(inputId = "num",
               label = "Number of variables",
               value = 100,
               min = 1),
  plotOutput(outputId = "hist")
)

# Define server function
server <- function(input, output) {

  data <- reactive({rexp(input$num)})

  output$hist <- renderPlot({
    ggplot(as_tibble(data()),
           aes(x = value)) +
      geom_histogram(bins = 10) +
      labs(title = input$title)
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

# Complementary functions for reactivity

Function	Purpose
<code>isolate()</code>	prevent reactions
<code>observeEvent()</code>	trigger code, useful for action button
<code>observe()</code>	similar to <code>observeEvent()</code>
<code>eventReactive()</code>	delay reactions

# Dynamic UI

# Dynamic interface

Shiny offers four main approaches to build a dynamic UI:

1. Function `conditionalPanel()`: wraps UI elements, does require very very minimal JavaScript knowledge
2. Function `renderUI()`: use in `server()` in conjunction with the `uiOutput()` function in `ui`, lets you generate calls to UI functions and make the results appear in a predetermined place in the UI.
3. Functions `insertUI()` and `removeUI()`: allow you to add or remove pieces of UI code
4. Use JavaScript directly

# References

1. Shiny. (2020). <https://shiny.rstudio.com/>.