Introduction to Shiny Part 2

Luke McGuinness

Twitter: @mcguinlu

Deptartment of Population Health Sciences,
Bristol Medical School

16th March, 2021

Overview of this session

- Background to `shiny`
- Getting started
  - Set-up
  - Control widgets & User interface
  - Outputs
- Getting more from `shiny`
  - Execution
  - Customising your app
  - Reactive programming
  - Publishing your app

New example!

Can anyone describe what this app does?

```r
library(shiny)
library(ggplot2)

med <- read.csv("http://bit.ly/bris-data-viz-med")

ui <- fluidPage(
      selectInput(inputId = "fill",
                  label = "Variable to fill by:",
                  choices = c("health","treatment")),

      plotOutput("barPlot")
)

server <- function(input, output) {

  output$barPlot <- renderPlot({

      ggplot(data = med, aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")

      })
}

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

New example!

Imports `med` dataset...

```r
library(shiny)
library(ggplot2)

med <- read.csv("http://bit.ly/bris-data-viz-med")

ui <- fluidPage(
      selectInput(inputId = "fill",
                  label = "Variable to fill by:",
                  choices = c("health","treatment")),

      plotOutput("barPlot")
)

server <- function(input, output) {

  output$barPlot <- renderPlot({

      ggplot(data = med, aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")

      })
}

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

New example!

Which is used to create a barplot using `ggplot2`...

```r
library(shiny)
library(ggplot2)

med <- read.csv("http://bit.ly/bris-data-viz-med")

ui <- fluidPage(
      selectInput(inputId = "fill",
                  label = "Variable to fill by:",
                  choices = c("health","treatment")),

      plotOutput("barPlot")
)

server <- function(input, output) {

  output$barPlot <- renderPlot({

      ggplot(data = med, aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")

      })
}

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

New example!

With user defined "fill" variable, captured by an input widget

```r
library(shiny)
library(ggplot2)

med <- read.csv("http://bit.ly/bris-data-viz-med")

ui <- fluidPage(
      selectInput(inputId = "fill",
                  label = "Variable to fill by:",
                  choices = c("health","treatment")),

      plotOutput("barPlot")
)

server <- function(input, output) {

  output$barPlot <- renderPlot({

      ggplot(data = med, aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")

      })
}

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

Getting more from `shiny`: Execution

Where you put your code is important

In the example, we loaded the `shiny` and `ggplot2` packages using:

```r
library(shiny)
library(ggplot2)
```

Due to the way your app is executed, it is important that this is put outside both the user interface container and the server

If you are `source()`-ing additional `.R` files, these commands should also go here

This is particularly relevant when loading large data files

When is your code run?

Only ever once, when the app is launched

```r
library(shiny)
library(ggplot2)

med <- read.csv("http://bit.ly/bris-data-viz-med")

ui <- fluidPage(
      selectInput(inputId = "fill",
                  label = "Variable to fill by:",
                  choices = c("health","treatment")),
      plotOutput("barPlot")
)

server <- function(input, output) {

   output$barPlot <- renderPlot({

      ggplot(data = med, aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")

      })
}

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

When is your code run?

Once each time a new user visits

```r
library(shiny)
library(ggplot2)

med <- read.csv("http://bit.ly/bris-data-viz-med")

ui <- fluidPage(
      selectInput(inputId = "fill",
                  label = "Variable to fill by:",
                  choices = c("health","treatment")),
      plotOutput("barPlot")
)

server <- function(input, output) {

   output$barPlot <- renderPlot({

      ggplot(data = med, aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")

      })
}

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

When is your code run?

Each time the value of `input$fill` changes (*reactive outputs*)

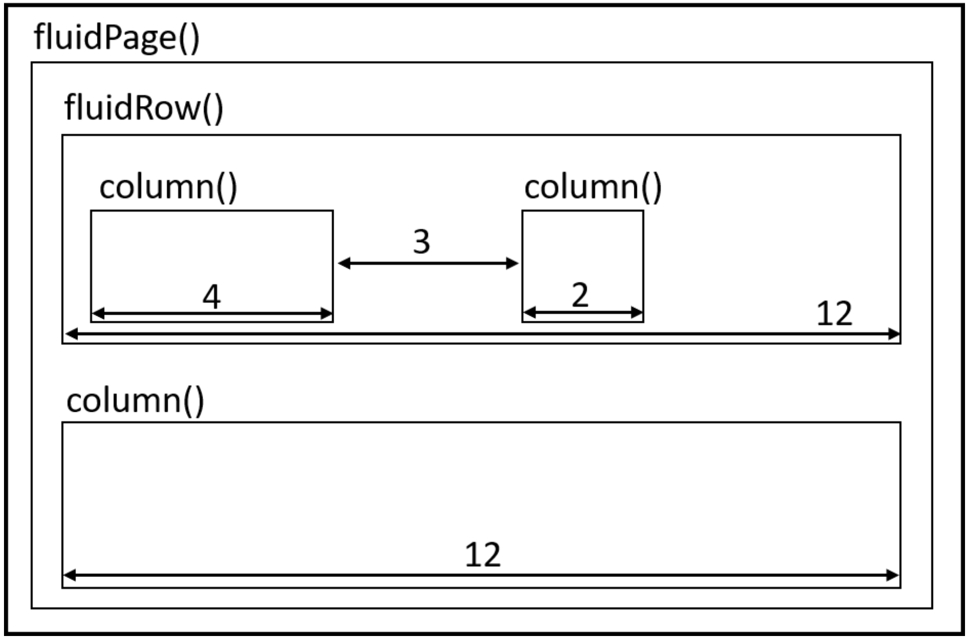
```r
library(shiny)
library(ggplot2)

med <- read.csv("http://bit.ly/bris-data-viz-med")

ui <- fluidPage(
      selectInput(inputId = "fill",
                  label = "Variable to fill by:",
                  choices = c("health","treatment")),
      plotOutput("barPlot")
)

server <- function(input, output) {

   output$barPlot <- renderPlot({

      ggplot(data = med, aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")

      })
}

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

Getting more from `shiny`: Page layout

Alternatives to `sidebarLayout()`

`fluidRow()`

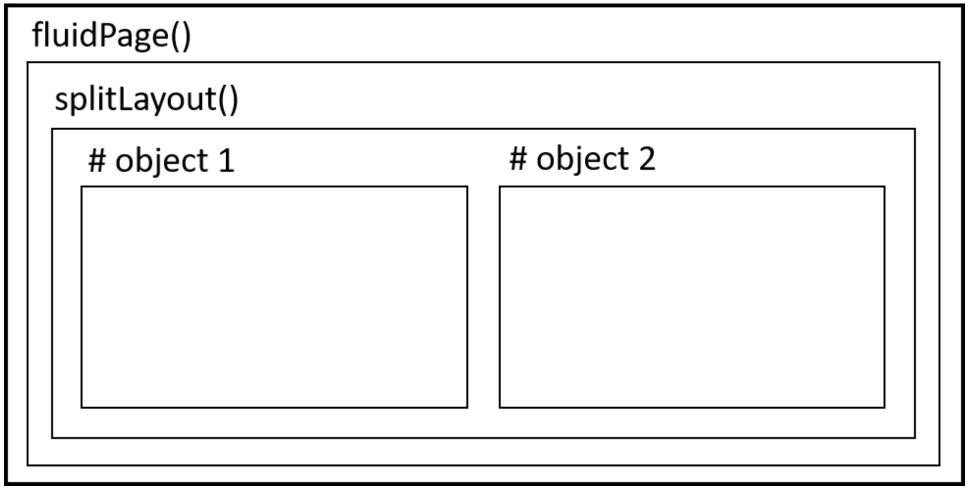
```
ui <- fluidPage(
 fluidRow(column(width= 4),
          column(width= 2), offset= 3),
 fluidRow(column(width= 12))
 )
```



Alternatives to `sidebarLayout()`

`splitLayout()`

```
ui <- fluidPage(
 splitLayout(
    # object 1,
    # object 2
 )
)
```



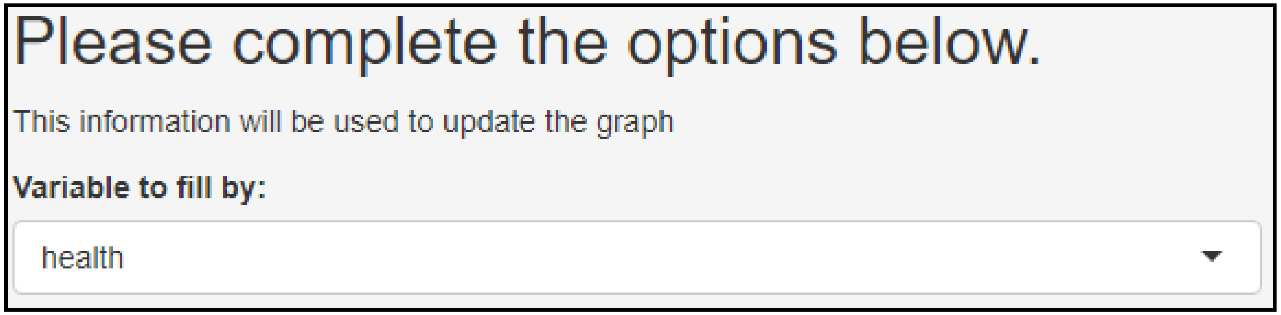
Getting more from `shiny`: Making `shiny` sparkle

A `shiny` UI is a HTML document

Can use HTML ***tags*** to add static elements (such as text) to your app.

For example, in the user interface:

```
ui <- fluidPage(
  h2("Please complete the options below."),
  p("This information will be used to update the graph"),
  selectInput(inputId = "fill",
              label = "Variable to fill by:",
              choices = c("health","treatment"))
)
```


Please complete the options below.

This information will be used to update the graph

Variable to fill by:

health

Adding HTML *tags*

The most common HTML tags have wrapper functions to make them easier to use:

```r
ui <- fluidPage(
 h1("Header 1"),                  # header (can be h1-h6)
 hr(),                            # horizontal rule
 br(),                            # line break
 p("Text"),                       # paragraph text
 p(strong("bold")),               # bold
 p(em("italic")),                 # italics
 p(code("code")),                 # code highlighting
 a(href="", "link"),              # Hyperlink
 HTML("<p>Raw html</p>")          # Raw html
)
```

Some tags, e.g. `em()` and `strong()` must be nested within a paragraph tag, `p()`

Similar to the control widgets, all tags are followed by a comma, except for the last element in the user interface container

A full list of tags is available on the `shiny` Rstudio cheatsheet

Add some text to our example app:

```
library(shiny)
library(ggplot2)

med <- read.csv("http://bit.ly/bris-data-viz-med")

ui <- fluidPage(
      p("Use the option below to customise your chart:"),
      selectInput(inputId = "fill",
                  label = "Variable to fill by:",
                  choices = c("health","treatment")),


      hr(),
      p("Here is your plot:"),
      plotOutput("barPlot")
)

server <- function(input, output) {

   output$barPlot <- renderPlot({

      ggplot(data = med, aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")

      })
}

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

Getting more from `shiny`: Reactive programming

New example!

```r
library(shiny)
library(ggplot2)
med <- read.csv("http://bit.ly/bris-data-viz-med")

ui <- fluidPage(
      selectInput(inputId = "fill",
                  label = "Fill:",
                  choices = c("health","treatment")),

      sliderInput(inputId = "slider",
                  label = "Number of rows to plot in figure:",
                  value = 100,
                  min = 5,
                  max = 150),

      plotOutput("barPlot")
)

server <- function(input, output) {

  output$barPlot <- renderPlot({
      med <- head(med, input$slider)

      ggplot(data = med, aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")
      })
}
```

Capture two user inputs: "fill" variable and number of rows to plot ("slider")

```r
library(shiny)
library(ggplot2)
med <- read.csv("http://bit.ly/bris-data-viz-med")

ui <- fluidPage(
      selectInput(inputId = "fill",
                  label = "Fill:",
                  choices = c("health","treatment")),

      sliderInput(inputId = "slider",
                  label = "Number of rows to plot in figure:",
                  value = 100,
                  min = 5,
                  max = 150),

      plotOutput("barPlot")
)

server <- function(input, output) {

  output$barPlot <- renderPlot({
      med <- head(med, input$slider)

      ggplot(data = med, aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")
      })
}
```

Use these inputs to create a barplot using the `med` dataset

```r
library(shiny)
library(ggplot2)
med <- read.csv("http://bit.ly/bris-data-viz-med")

ui <- fluidPage(
      selectInput(inputId = "fill",
                  label = "Fill:",
                  choices = c("health","treatment")),

      sliderInput(inputId = "slider",
                  label = "Number of rows to plot in figure:",
                  value = 100,
                  min = 5,
                  max = 150),

      plotOutput("barPlot")
)

server <- function(input, output) {

  output$barPlot <- renderPlot({
      med <- head(med, input$slider)

      ggplot(data = med, aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")
      })
}
```

What does the output barPlot take a dependency on?

```r
library(shiny)
library(ggplot2)
med <- read.csv("http://bit.ly/bris-data-viz-med")

ui <- fluidPage(
      selectInput(inputId = "fill",
                  label = "Fill:",
                  choices = c("health","treatment")),

      sliderInput(inputId = "slider",
                  label = "Number of rows to plot in figure:",
                  value = 100,
                  min = 5,
                  max = 150),

      plotOutput("barPlot")
)

server <- function(input, output) {

  output$barPlot <- renderPlot({
      med <- head(med, input$slider)

      ggplot(data = med, aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")
      })
}
```

Shiny is reactive . . .

`output$barplot` depends on the value of **both** `input$slider` and `input$fill`:

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

 output$barPlot <- renderPlot({
      data <- head(med, n = input$slider)

      ggplot(data = data, aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")
  })
}
```

When either `input$slider` and `input$fill` changes, all the code contain within the `renderPlot()` function will run

. . . but in a lazy way

Compare what happens when we change `input$fill` in the following:

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

 output$barPlot <- renderPlot({
      data <- head(med, n = input$slider)

      ggplot(data = data, aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")
  })}
```

versus:

```r
server <- function(input, output) {
  data <- reactive({
        head(med, n = input$slider)
  })

 output$barPlot <- renderPlot({
    ggplot(data = data(), aes_string(fill = input$fill)) +
      geom_histogram(aes(x = status), stat = "count")
  })}
```

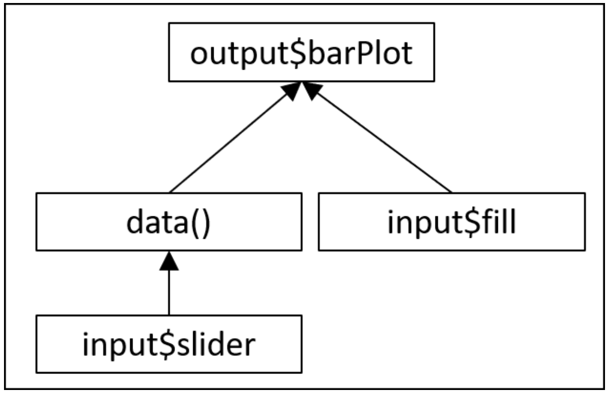
The `reactive({ })` function

The difference: changing the value of `input$fill` in the second example does not require the data to be re-calculated:

- `renderPlot` calls `data()`

```
data <- reactive({ head(med, n = input$slider) })
```

- `data()` will check that the value of `input$slider` has not changed
- `data()` will return its saved data ***without*** re-subsetting the *med* dataset
- `renderPlot` will re-draw the histogram with the correct fill.



The `reactive({ })` function

Shiny caches the results of `data()` and continuously validates the value of `input$slider` on which it depends.

Being able to separate out computationally intense steps in your app is useful, as it prevents Shiny from re-running code unnecessarily.

For example, reloading and cleaning a large datasets from the web each time the user makes a change to a plot title is inefficient

Getting more from `shiny`: Sharing/publishing your app

Sharing/publishing your app

At the moment, your app is only available to you locally

To make it widely available, you need to publish it

Lots of ways to do this, but easiest is via shinyapps.io:

- Ensure that your app.R file is contained within its own folder
- Go to shinyapps.io
- Follow the instructions there to publish your app

Sharing/publishing your app

Why publish?

- Makes your code available to anyone with an internet connection
- Good way to showcase your work
- Can make a nice compliment to an R package, as it does not require users to know R

Wrapping up

While the apps we built today are quite simple, the possibilities with `shiny` are endless:

Example 1 Example 2

Please do get in touch if I can be of help:

- Email: luke.mcguinness@bristol.ac.uk
- Twitter: @mcguinlu
- GitHub: @mcguinlu