

Introduction to Shiny

Luke McGuinness

Twitter: @mcguinlu

**Department of Population Health Sciences,
Bristol Medical School**

15th November, 2019

Overview of this session

- Background to shiny
- Creating your own app
 - Set-up
 - User interface & control widgets
 - Outputs
- Getting more from shiny
 - Execution
 - Customising your app
 - Reactive programming
 - Publishing your app

Public Service Announcements

Workshop plan

- Lecture and six short practicals mixed together
- There will be a break for coffee in the middle

Please ask questions as we go along, and feel free to chat/compare code with your neighbours

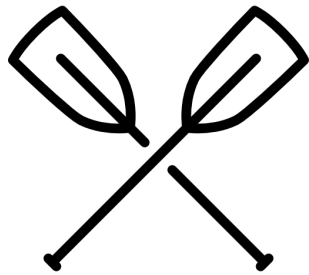
We will be using timers as part of this course:



00:15

Public Service Announcements

I pronounce the letter "R" oddly:



R
("oar")

means



R
("arr")

▲ Why are open source statistical programming languages the best?
46 Because they R.
▼ share cite edit answered Jul 8 '13 at 15:29
add a comment

Introduction: Background to shiny

What is shiny?

shiny is an R package that allows users to build interactive web applications ("apps") straight from R.



Why is it called shiny?

Likely (but less interesting) reason

Allows you “*to build beautiful, responsive, and powerful applications with minimal effort.*”

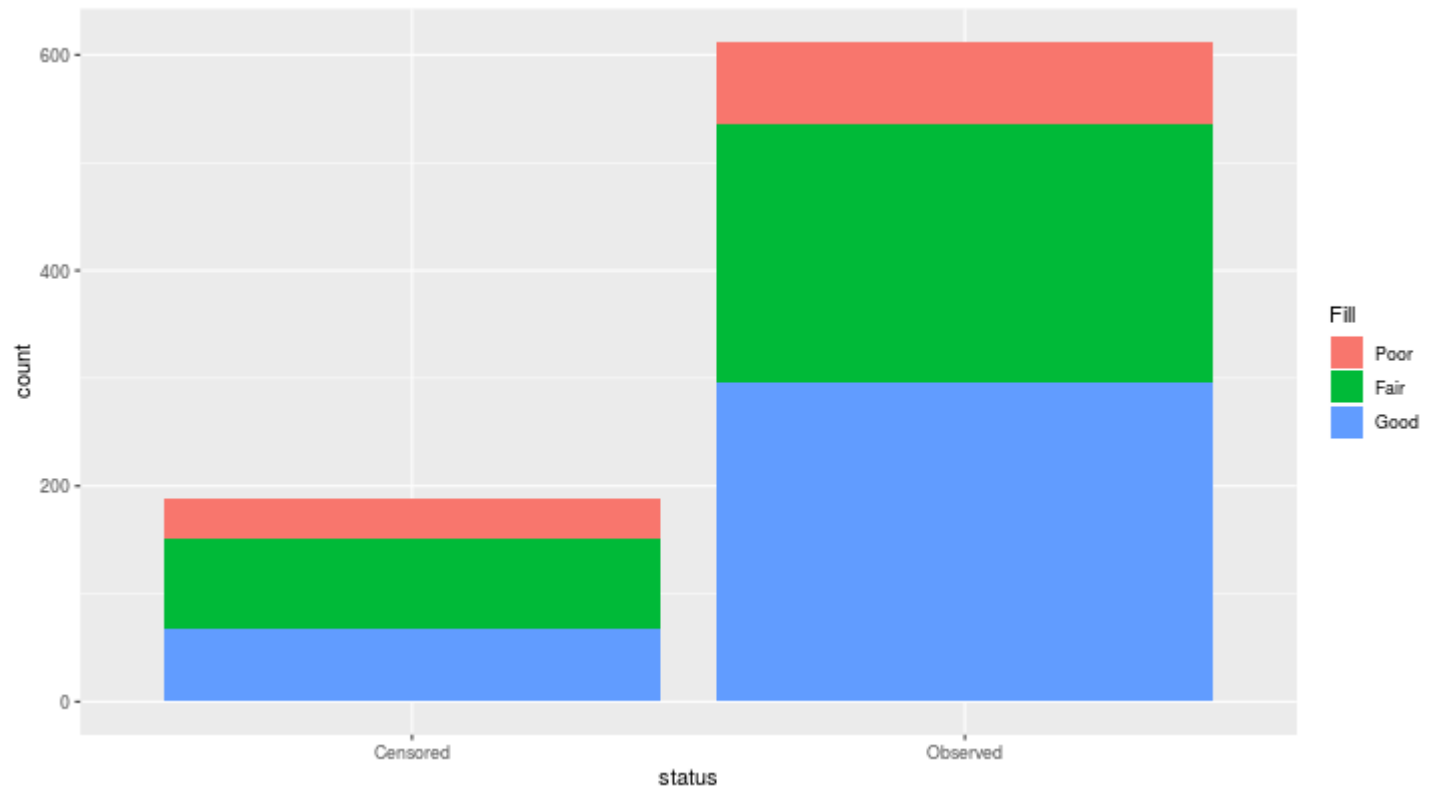
Unlikely (but more interesting) reason

Speculation that it comes from the show *Firefly*, in which “shiny” means “good” or “valuable”.



Benefits of shiny for visualisation

Variable to fill by:



Data shown is from the med dataset used throughout this course

Introduction: Structure of a shiny app

Structure of a shiny app

A shiny app has three components:

- **User interface**

- Defines the layout of your app
- Controls what it looks like (themes/fonts/etc)

- **Server**

- Defines the logic needed to build the app
- Performs computational work

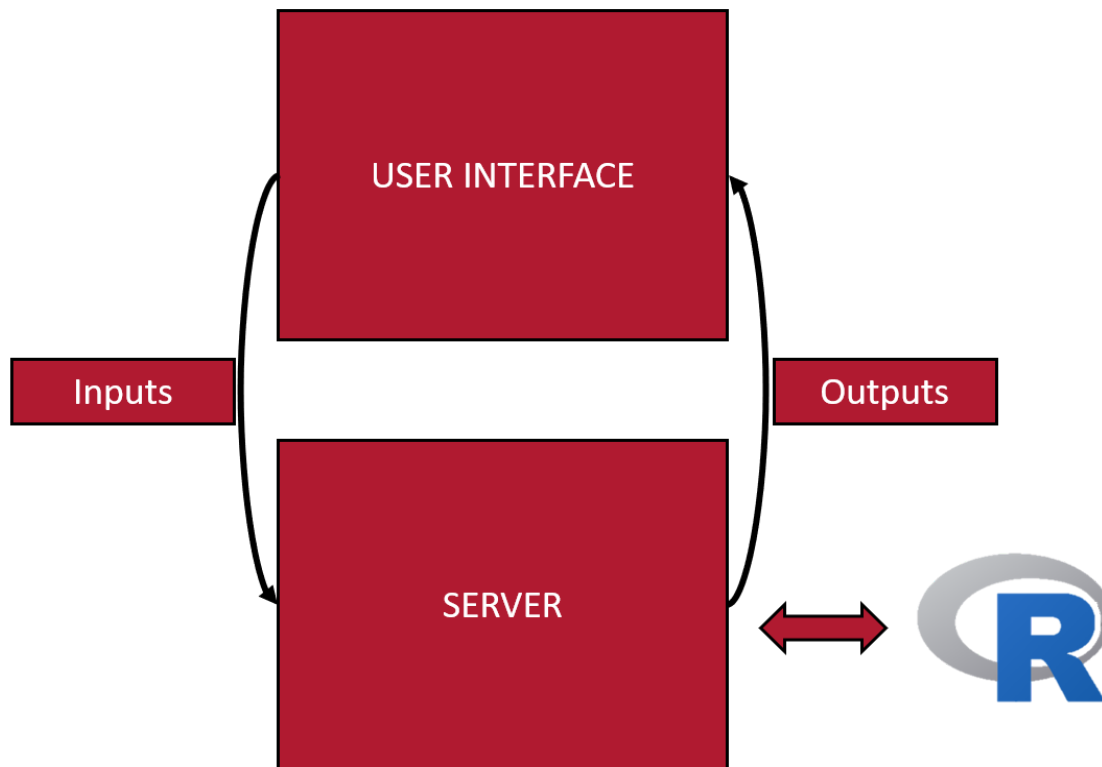
- **Call to the shinyApp function**

- Creates the app from an user interface and server pair

Inputs and outputs

Information moves between the UI and the server via inputs and outputs

NB: No computation takes place in UI - only the server can run R functions



Creating your own app: Getting started

Setting up a shiny app

This *ui/server* structure can be implemented in two ways:

Option 1

A single file called `app.R`

Option 2

Two files: `ui.R` and `server.R`



**This is the structure
we will use.**

Option 1: single file (*app.R*)

User interface container

```
# USER INTERFACE
```

```
ui <- fluidPage(  
  . . .  
)
```

```
# SERVER  
|  
|  
|
```

```
# RUN THE APP  
|
```

Option 1: single file (*app.R*)

Server function

```
# USER INTERFACE
ui <- fluidPage(
  . . .
)

# SERVER
server <- function(input, output) {
  . . .
}

# RUN THE APP
|
```

Option 1: single file (*app.R*)

Call to the `shinyApp` function

```
# USER INTERFACE
ui <- fluidPage(
  . . .
)

# SERVER
server <- function(input, output) {
  . . .
}

# RUN THE APP
shinyApp(ui = ui, server = server)
```


Option 2: two-file approach

Create new folder named "app_directory" with two files:

```
~/app_directory  
|-- ui.R  
|-- server.R
```

The files will contain the relevant functions:

```
ui<-fluidPage(  
  . . .  
)
```

ui.R

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

server.R

Then run the app using:

```
runApp("app_directory")
```

Running your app

There are two options when running your app:

Open the app by running `runApp("app-dir")` in the console, where "app-dir" is the name of the directory containing your app.R script:

```
runApp("luke")
```

OR

Open the app.R script in Rstudio and then:

- click the "Run App" button:



- Use the keyboard short-cut: *Ctrl/Command+Shift+Enter*

Your Turn: Introducing the data

The dataset we will use for the practical elements is the `bmi2` dataset included with the `BristolVis` package:

id	age	bmi	sex	diet	status
1	78.2	29.3	Male	Good	Unhealthy
2	48.5	33.0	Female	Good	Unhealthy
3	79.5	31.5	Female	Good	Unhealthy
4	78.5	28.1	Male	Poor	Healthy

The data set contains 200 observations across the following six variables:

Continuous variables:

- `id`
- `age`
- `bmi`

Categorical variables:

- `sex`: Male / Female
- `diet`: Good / Moderate / Poor
- `status`: Healthy / Unhealthy

Your Turn: Run your first app

- Open the "prac1_shiny" vignette, and copy the code into an empty R file

```
vignette("prac1_shiny", package = "BristolVis")
```

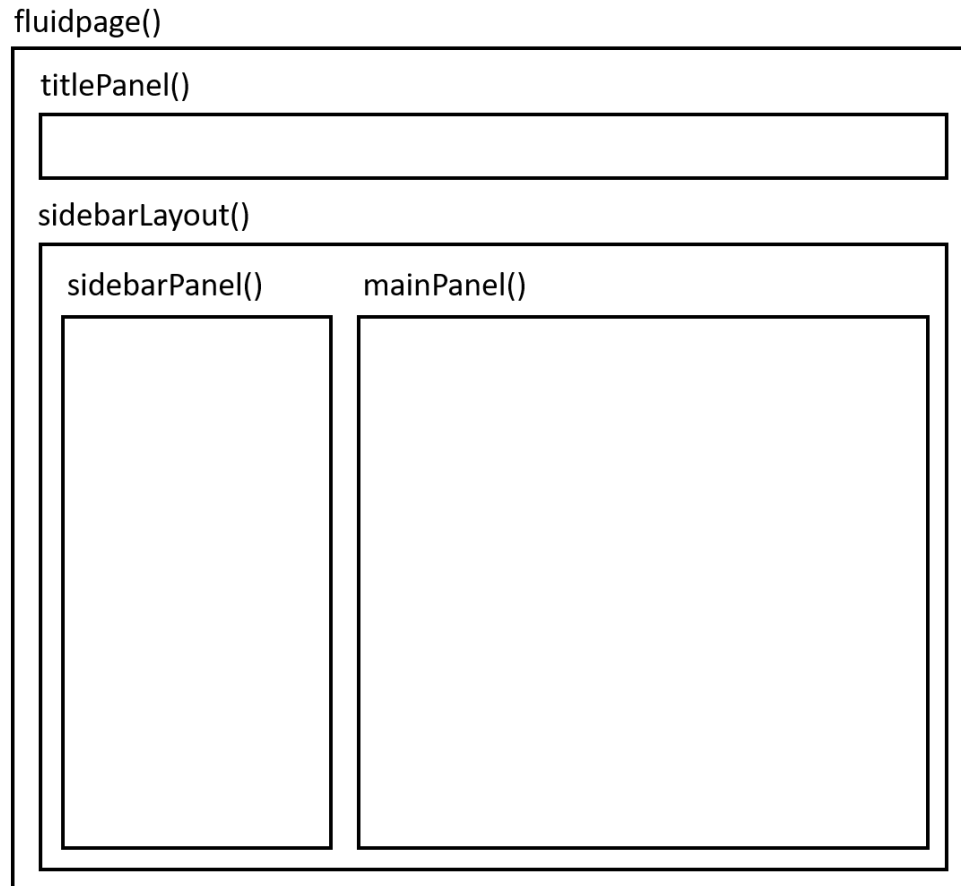
- Run it both:
 - from the console (note: you'll have to save it first)
 - using the "Run App" button

05:00

**Creating your own app:
User interface &
control widgets**

sidebarLayout()

Common layout for shiny apps



sidebarLayout()

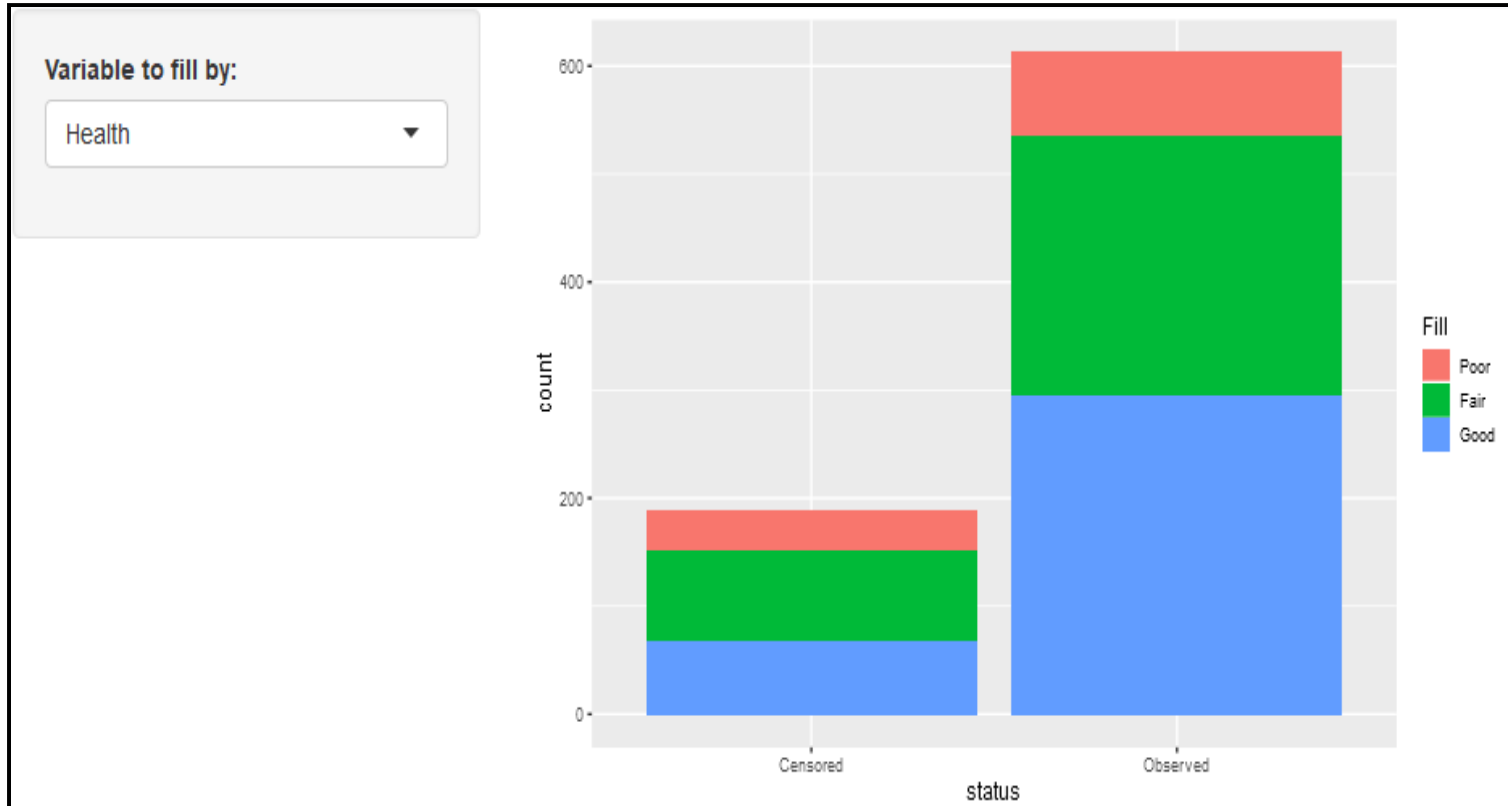
Defined in code as:

```
ui <- fluidPage(  
  sidebarLayout(  
    sidebarPanel(  
      # Control widgets and HTML objects  
    ),  
    mainPanel(  
      # Outputs  
    )  
  )  
)
```

Conventions for use:

- The `sidebarPanel` is usually used to house the control widgets that capture user input.
- The `mainPanel` is usually used to present the output of the app (text/graph/results).

Example of the sidebarLayout()



A widget for every occasion

Control widgets are used to capture user input and vary based on type of input:


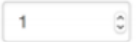






	actionButton (inputId, label, icon, ...)		numericInput (inputId, label, value, min, max, step)
Link	actionLink (inputId, label, icon, ...)		passwordInput (inputId, label, value)
<input checked="" type="checkbox"/> Choice 1 <input checked="" type="checkbox"/> Choice 2 <input type="checkbox"/> Choice 3	checkboxGroupInput (inputId, label, choices, selected, inline)	<input checked="" type="radio"/> Choice A <input type="radio"/> Choice B <input type="radio"/> Choice C	radioButtons (inputId, label, choices, selected, inline)
<input checked="" type="checkbox"/> Check me	checkboxInput (inputId, label, value)	<input type="text" value="Choice 1"/> ^	selectInput (inputId, label, choices, selected, multiple, selectize, width, size) (also selectizeInput())
	dateInput (inputId, label, value, min, max, format, startview, weekstart, language)	<input type="text" value="Choice 1"/> <input type="text" value="Choice 2"/>	
	dateRangeInput (inputId, label, start, end, min, max, format, startview, weekstart, language, separator)		sliderInput (inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)
	fileInput (inputId, label, multiple, accept)		submitButton (text, icon) (Prevents reactions across entire app)
		<input type="text" value="Enter text"/>	textInput (inputId, label, value)

Image sourced from the [shiny Rstudio cheatsheet](#)

The anatomy of a widget

All control widgets have two elements in common:

- **inputId**: Unique ID for that widget
- **label**: Text to be displayed beside the widget (which can be left blank)

```
textInput(inputId = "textboxinput",  
          label = "")
```



Other elements are specific to the widget you are using:

```
selectInput(inputId = "fillby",  
            label = "Variable to fill by",  
            choices = c("Health", "treatment")),
```



A note on commas in the user interface

Remember that the UI containers (e.g. `mainPanel()`) are still functions, to which arguments (i.e. widgets) are passed, so normal rules apply when using commas

Your Turn: Design a user interface

- Open the "prac1_shiny" vignette, and copy the code into an empty R file

```
vignette("prac1_shiny", package = "BristolVis")
```

- Identify the elements of `sidebarLayout`
- Run the app
- Make the following changes:
 - Change the title of the app to "Demo of a shiny app"
 - Add a new `selectInput` widget to the sidebar, with the options "sex", "diet", and "status"
 - Add a new `textInput` widget to the sidebar with the label "Plot title:"
- What is the value of `input$textbox` when the following is entered in it?
This is a test

10:00

Creating your own app: Reactive outputs

Creating outputs - the basics

```
# Define UI
ui <- fluidPage(
  textOutput("textout")
)

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

  output$textout <- renderText({
    "This is the best app ever!"
  })
}

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

Two steps:

- Add the code to render the object to the server function using a `render*()` function

Creating outputs - the basics

```
# Define UI
ui <- fluidPage(
  textOutput("textout")
)

# Define server
server <- function(input, output) {
  output$textout <- renderText({
    "This is the best app ever!"
  })
}

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

Two steps:

- Add the code to render the object to the server function using a `render*()` function
- Add the resulting object to the user interface using the corresponding `*Output()` function.

Rendering the output

Similar to inputs, there are different `render*()/Output()` function pairs for different types of output:

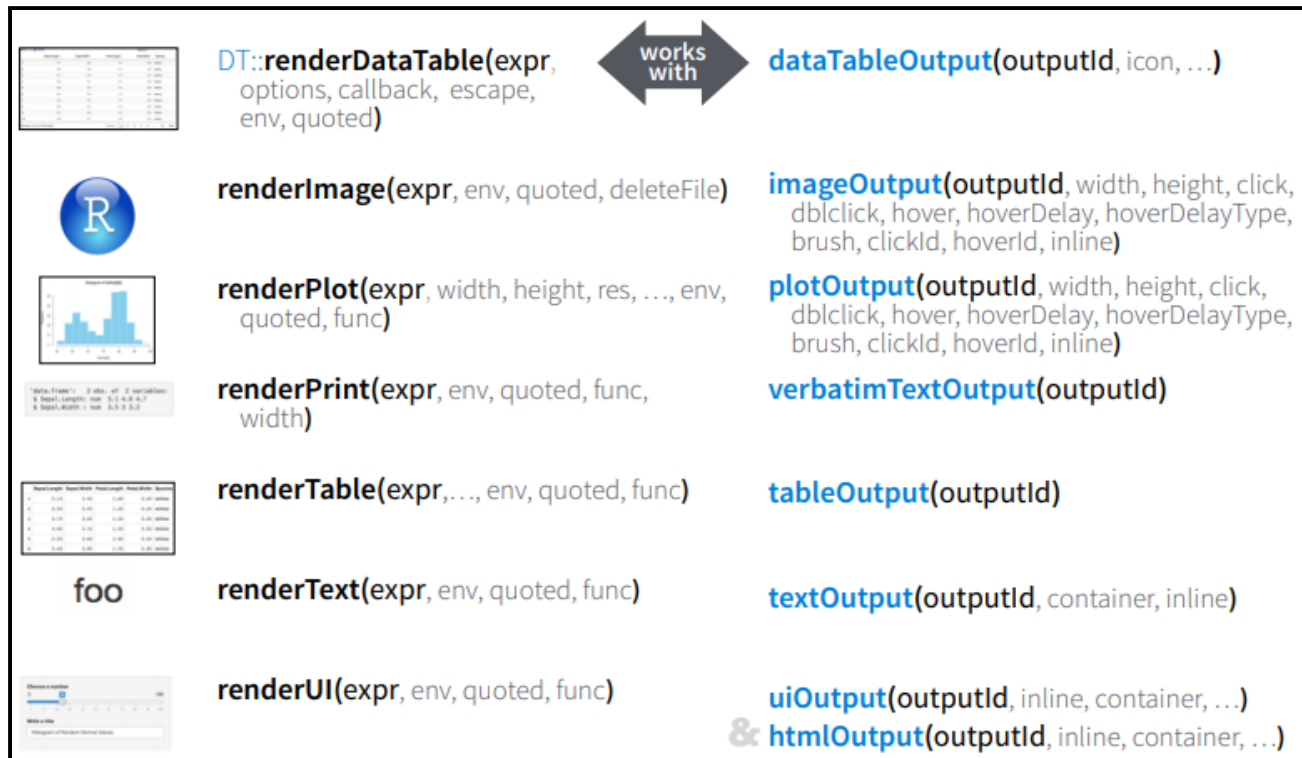


Image sourced from the [shiny Rstudio cheatsheet](#)

Static vs reactive outputs

So far, the graphical outputs we have created are not fully reactive

Remember in the last practical, changing the value of the selectInput widget from "sex" to "diet" had no effect

Reactive outputs are those that make use of the value of a control widget

Accessing widget values

The `inputId` of the widget is important

Remember the server function:

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

Reads in a list-like object containing the value of each widget, identified by its `inputId`

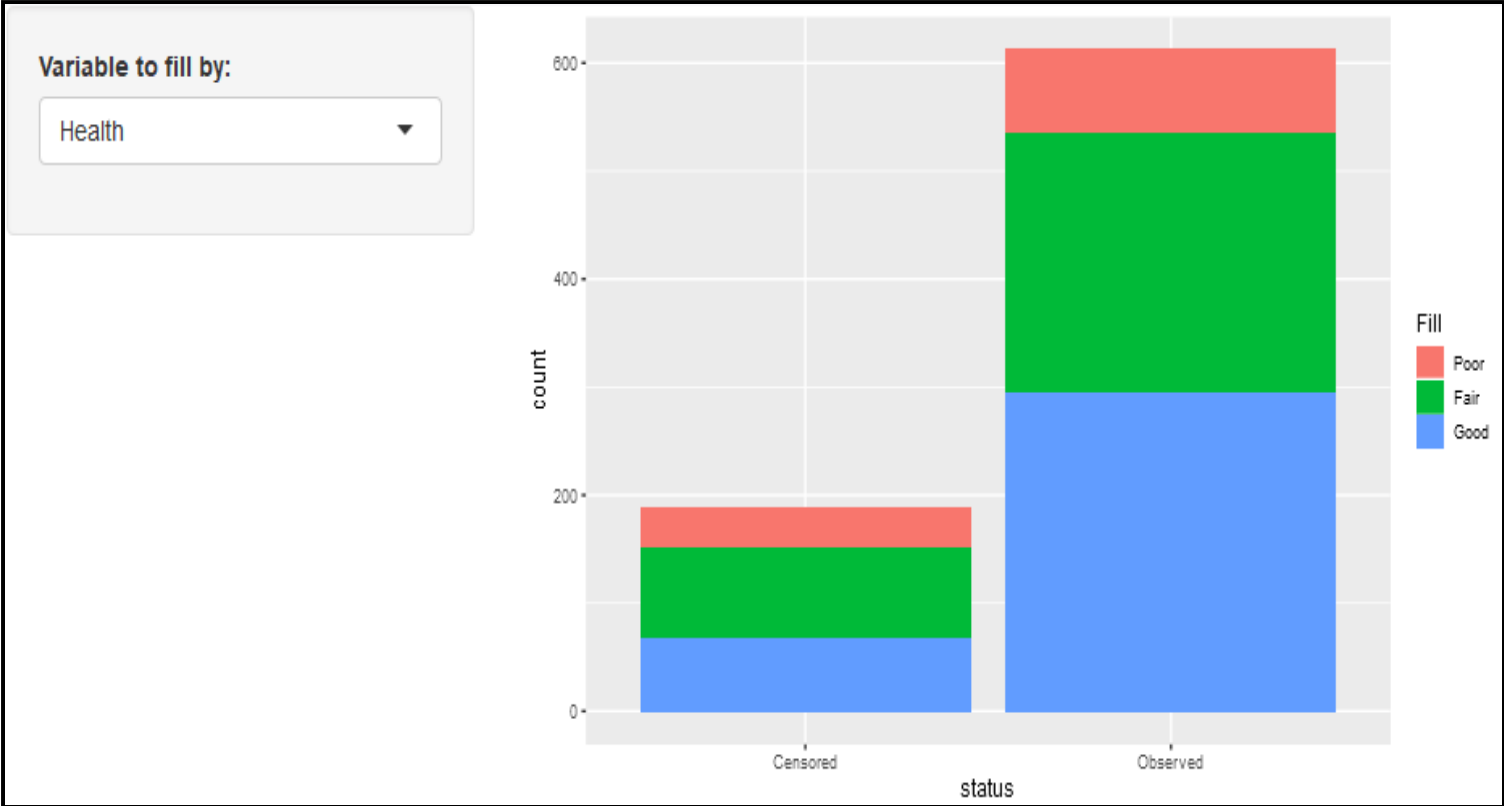
For the following widget:

```
textInput(inputId = "textboxinput",  
          label = "")
```

its value is defined by

```
input$textboxinput
```

Using the value of the control widgets



Using the value of the control widgets

renderPlot() function in the server

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

Changes based on value selected by user in UI

When *input\$fill = health*:

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

When *input\$fill = treatment*:

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

... and switch!

In the previous example, we used the `aes_string()` function to get around the fact that the value of `input$fill` was in quotes

The `switch` function converts the value of `input$<id>` from a `selectInput()` box into anything you like.

```
server <- function(input, output) {  
  output$barPlot <- renderPlot({  
    fill <- switch(input$fill,  
                  "treatment" = med$treatment,  
                  "health" = med$health)  
    ggplot(data = med, aes(fill = fill)) +  
      geom_histogram(aes(x = status), stat = "count")  
  })  
}
```

A comment on brackets

Note the difference between:

```
renderPlot({})
```

```
renderPlot()
```

Whenever you have a function argument that's more than just a single expression, you need to use curly braces

Good practice to use `{}` all the time, so that you don't get errors

Your Turn: Create reactive content

- Open the "prac2_shiny" vignette, and copy the code into an empty R file

```
vignette("prac2_shiny", package = "BristolVis")
```

- Run the app to familiarise yourself with it
- Examine the code used to produce the table in the UI/server
 - Why is the table not being displayed in the app?
 - Fix the code in the UI so that the table is displayed in the `mainPanel`
- Examine the code used to produce the plot in the UI/server
 - Update the code so that the colour is set by the value of `input$pointcolour`
 - Update the code so that the title of the plot is defined by the value of the input from the `inputTextbox` widget
- Re-run the app to ensure that the changes have worked

10:00

Getting more from shiny: Execution

Where you put your code is important

In "prac2_shiny", we loaded the `shiny`, `BristolVis` and `ggplot2` packages using:

```
library(shiny)
library(ggplot2)
library(BristolVis)
```

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

When is your code run?

Only ever once, when the app is launched

```
library(shiny)
library(BristolVis)
library(ggplot2)

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

```
library(shiny)
library(BristolVis)
library(ggplot2)

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

```
library(shiny)
library(BristolVis)
library(ggplot2)

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")
  })
}

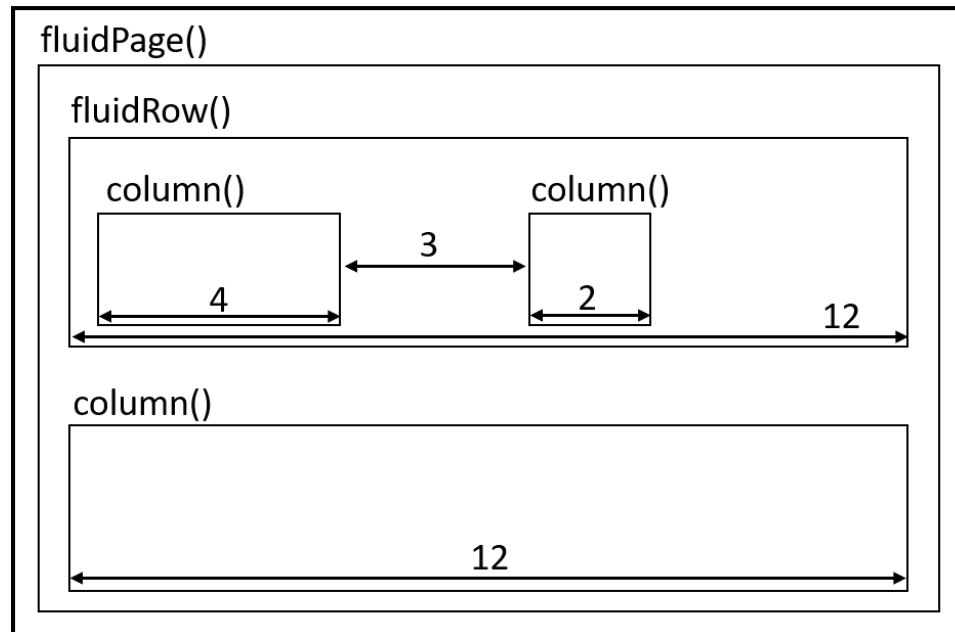
shinyApp(ui = ui, server = server)
```

Getting more from shiny: Page layout

Alternatives to sidebarLayout()

fluidRow()

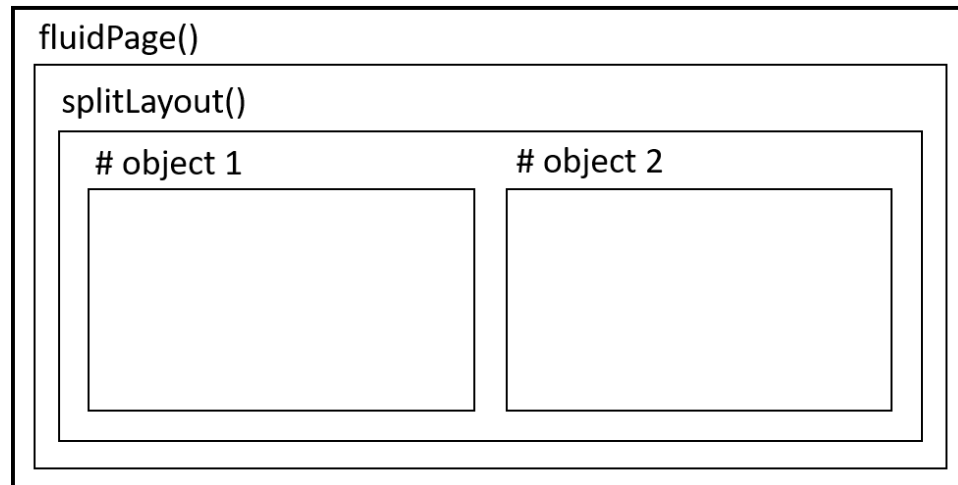
```
ui <- fluidPage(  
  fluidRow(column(width= 4),  
           column(width= 2), offset= 3),  
  fluidRow(column(width= 12))  
)
```



Alternatives to sidebarLayout()

splitLayout()

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



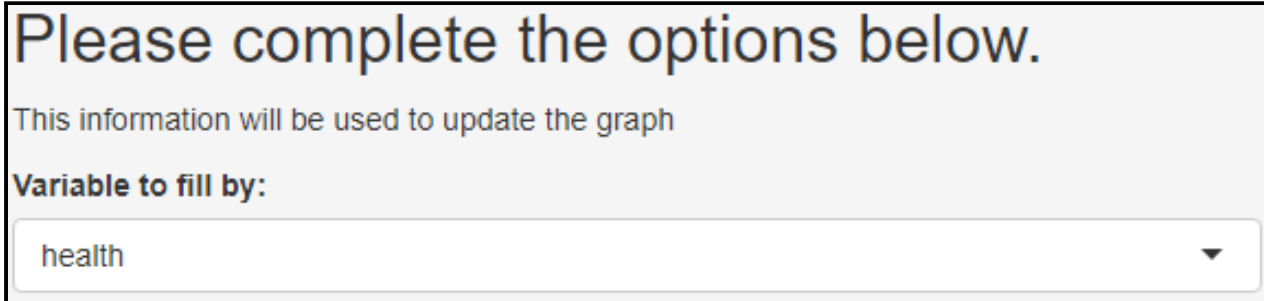
Getting more from shiny: Making shiny sparkle

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 tags have wrapper functions to make them easier to use:

```
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](#)

Getting more from shiny: Reactive programming

Shiny is reactive . . .

As discussed, `output$barplot` below depends on the value of **both** `input$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

02:00

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

```
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:

```
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")  
  })  
}
```

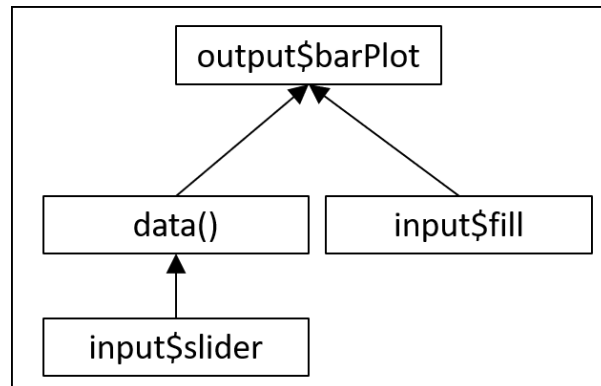
The `reactive({ })` function

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.



Your Turn: Reactive expressions & HTML tags

- Open the "prac3_shiny" vignette, and copy the code into an empty R file

```
vignette("prac3_shiny", package = "BristolVis")
```

- Run the app to see what it does
- Add some text to the end of the sidebar to describe the dataset used
- Add some text to the mainPanel() e.g. between the plot and the table
- Create a reactive expression so that the data does not have to be imported everytime the title is changed
- Re-run your app to ensure that it works as expected

Your Turn: Build an app from scratch

Open a new Shiny app (File -> New File -> Shiny Web App. . .)

Develop an app to create reactive graph

- Use one of the layouts we discussed as part of this workshop
- Load `shiny`, `BristolVis` and `ggplot2` packages
- Use one of the other datasets included with the `BristolVis` package
- Add some text to the app to describe the dataset used
- Add at least one widget to the `sidebarPanel` (but feel free to add as many as you like!)
- Add a plot to the `mainPanel` that makes use of the input from your widget

Run the app to ensure that it works as expected, and compare your finished app/code with a neighbour

30:00

Getting more from shiny: Sharing/publishing your app

Your Turn: Publish your app

- Ensure that your app.R file is contained within its own folder
- Deploy your app online
 - Go to shinyapps.io
 - Follow the instructions there to publish your app
 - Send the link to your new app, plus a brief description of what it does, to us here:

forms.gle/vR1MfqsfWNU4GBGj7

10:00

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](https://twitter.com/mcguinlu)

Images

Oar: By Florian P epellin - Own work, CC BY-SA 3.0,
<https://commons.wikimedia.org/w/index.php?curid=43042720>