Oct 2024

R Shiny Masterclass Series - Introduction

R Shiny widgets, reactivity and debugging

EPI-interactive

Agenda

- **Session 1** | 30 September | Getting started with Posit Cloud and your first R Shiny app
- **Session 2** | 01 October | R Shiny core concepts and mobile ready layout
- **Session 3** | 03 October | R Shiny user interface components, reactivity and debugging
- **Session 4** | 07 October | Data sources and data processing in R Shiny
- **Session 5** | 08 October | Interactive charts with Plotly: chart types, customising hover boxes and chart styling
- **Session 6** | 10 October | Maps and spatial visualisation with Leaflet: adding map layers, annotations, pins, filters and legend
- **Session 7** | 14 October | Publishing R Shiny apps, design considerations and case study
- **Session 8** | 15 October | Case study, top 10 tips for data visualisation with R Shiny and wrap-up

Today

Goals:

- Getting familiar with Shiny inputs and outputs
- Understand basics of reactivity
- Learn debugging tools (if we have time)

Steps:

- Add sliders, checkboxes, input fields etc.
- Use reactive expressions for calculations
- Create outputs to display results in the app

Reactive inputs and outputs

Reactive inputs

- UI components (widgets) that allow the user to interact with the app
- Generally found in the ui.R file
- Examples: sliderInput, dateInput, selectInput, checkboxInput, textInput

Reactive inputs in the UI

Check the Shiny widget gallery:

http://shiny.rstudio.com/gallery/widget-gallery.html

Check the code, e.g. for the slider input

```
sliderInput("id", "Label", min=1, max=5, value=3)
```

Arguments

More info: https://shiny.rstudio.com/reference/shiny/latest/

EPI-interactive

Reactive inputs in the Server

- input$id
    - Can also call using input$`id with spaces` or input[[“id.with.symbols”]]
- This value is read-only
- Updates when the user input changes
- Must be used in a reactive context*

*We will come back to this later

Reactive outputs

- Displays some visualisations or contextual information in app
- Usually based on calculations involving user inputs
- Examples: tableOutput, textOutput, uiOutput, plotOutput
- Needs to be declared in both in ui.R and server.R

Example:

- **textOutput(textID)** in ui.R
- **output$textID** <- renderText(…) in server.R

Reactive inputs and outputs - exercise

In Session-3/stage1

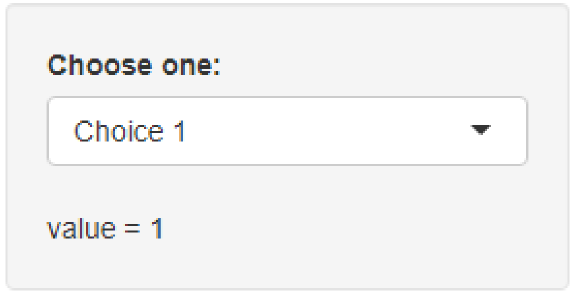
- Add a selectInput widget
- Add a textOutput that displays “value = [input value]”

  Hint: use paste() or paste0() to combine strings

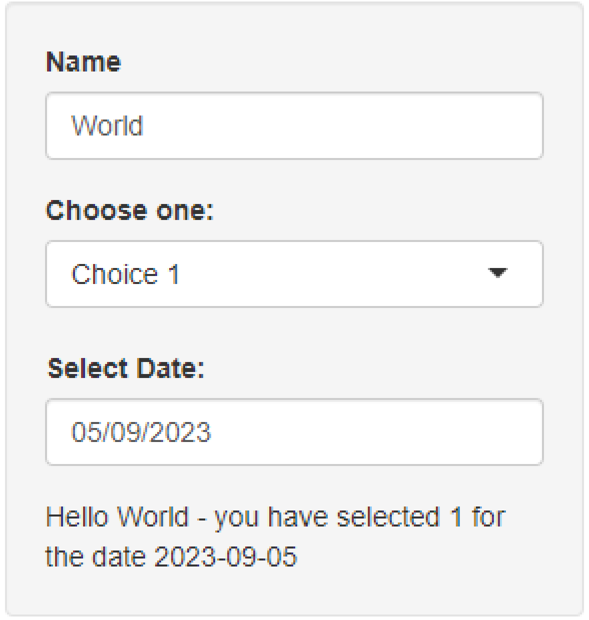
Extended exercise

- Add a textInput widget, a dateInput widget and an additional input widget of your choice

  http://shiny.rstudio.com/gallery/widget-gallery.html
- Include the new inputs in the textOutput



value = 1



Hello World - you have selected 1 for the date 2023-09-05

EPI-interactive

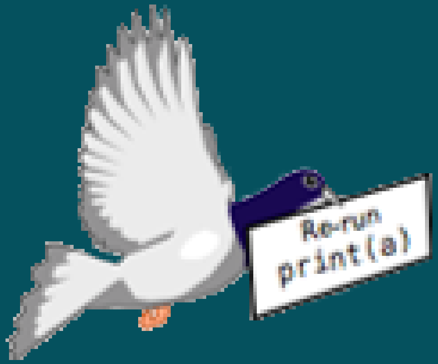
Reactive inputs and outputs - exercise

```
ui <- fluidPage(
    selectInput("select",
        "Choose one:",
        choices = list("Choice 1" = 1, "Choice 2" = 2, "Choice 3" = 3)),
    textOutput("display")
)

server <- function(input, output) {
    output$display <- renderText({
        paste("value = ", input$select)
    })
}
```

EPI-interactive

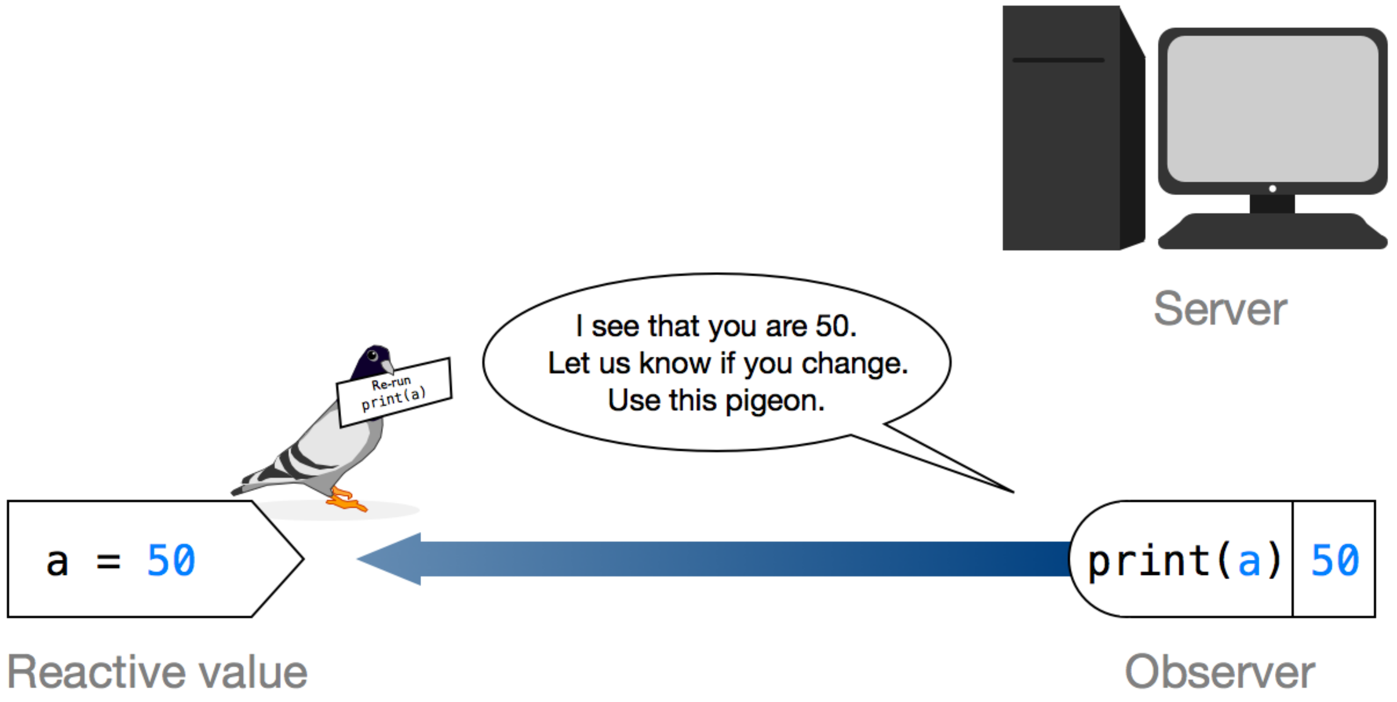
Reactive inputs and outputs - extended

```
ui <- fluidPage(
    textInput("selectText", label = "Name", value = "World"),
    selectInput("select",
        "Choose one:",
        choices = list("Choice 1" = 1, "Choice 2" = 2, "Choice 3" = 3)),
    dateInput("selectDate","Select Date:"),
    textOutput("display")
)

server <- function(input, output) {
    output$display <- renderText({
        paste("Hello ", input$selectText, "- you have selected choice number ", input$select," for the date ", input$selectDate)
    })
}
```



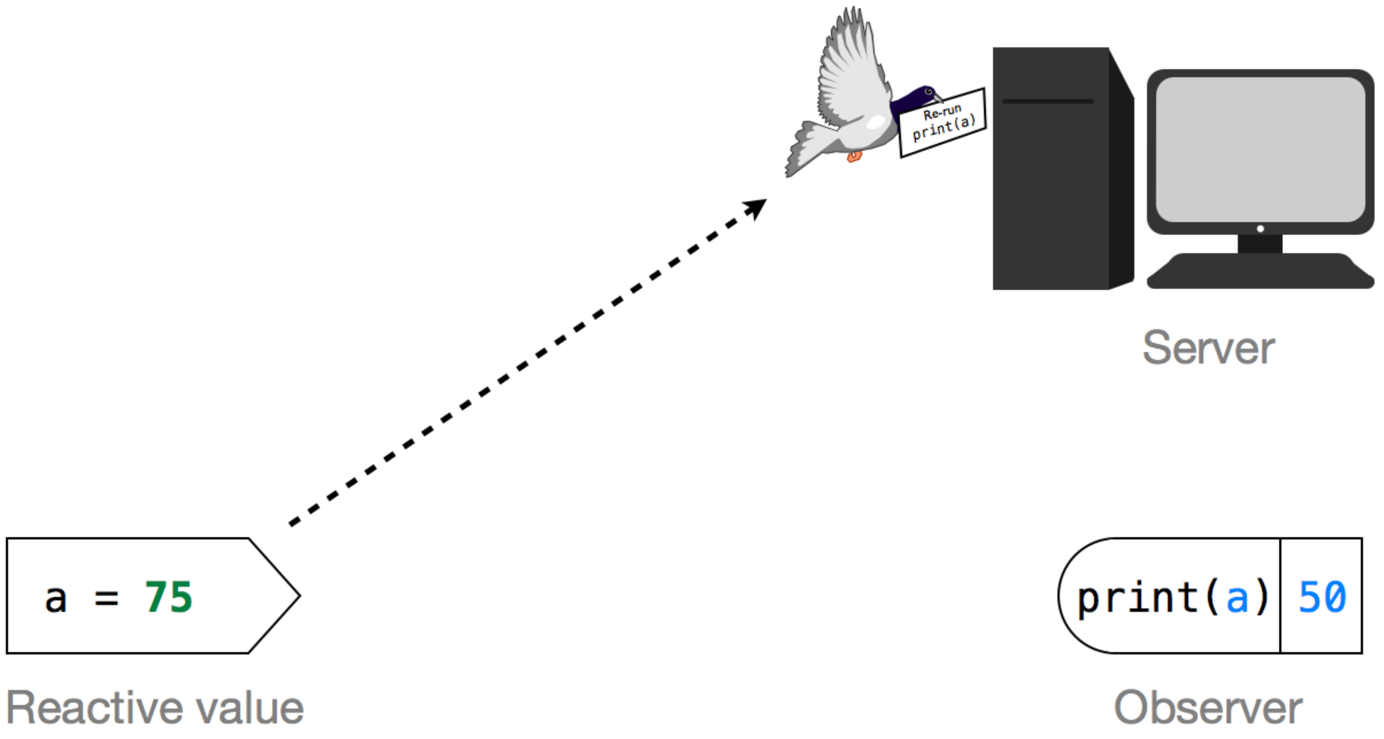
Reactivity

Reactivity Explained



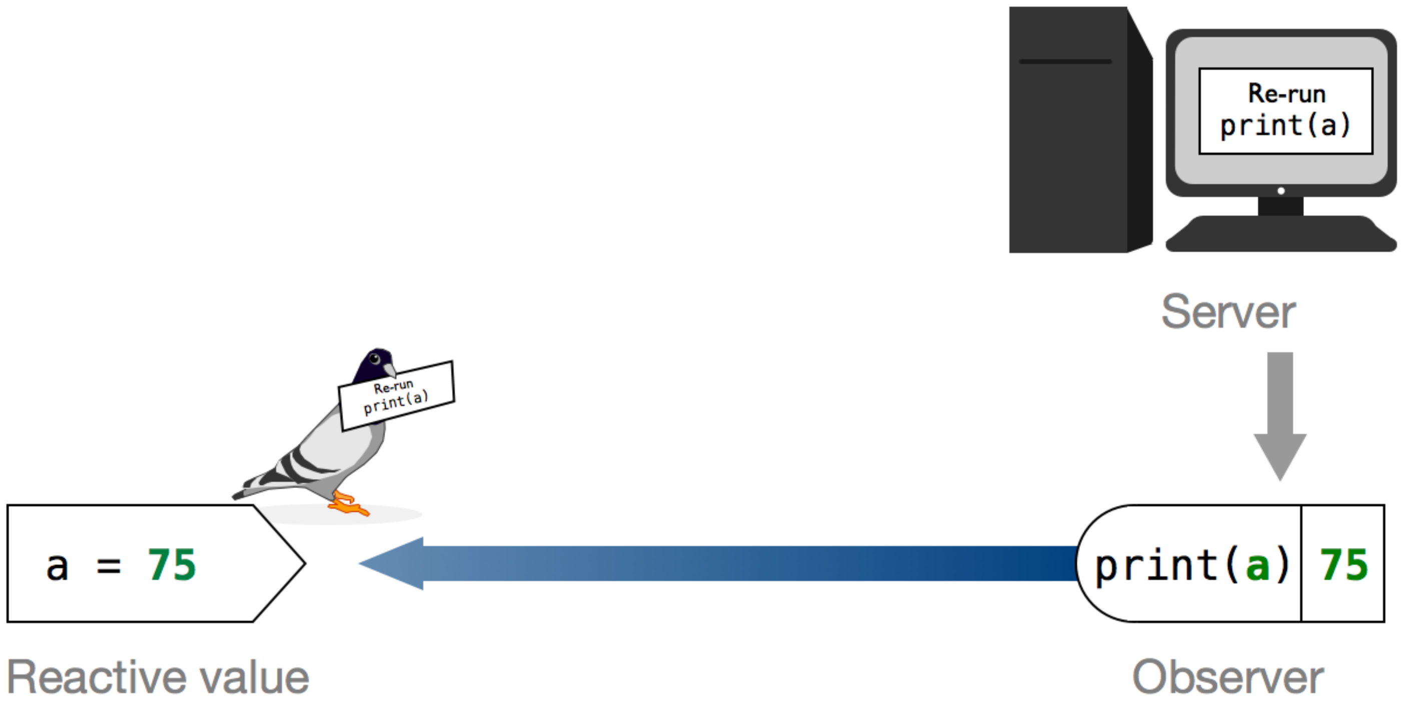
https://shiny.rstudio.com/articles/understanding-reactivity.html

Reactivity Explained



https://shiny.rstudio.com/articles/understanding-reactivity.html

Reactivity Explained

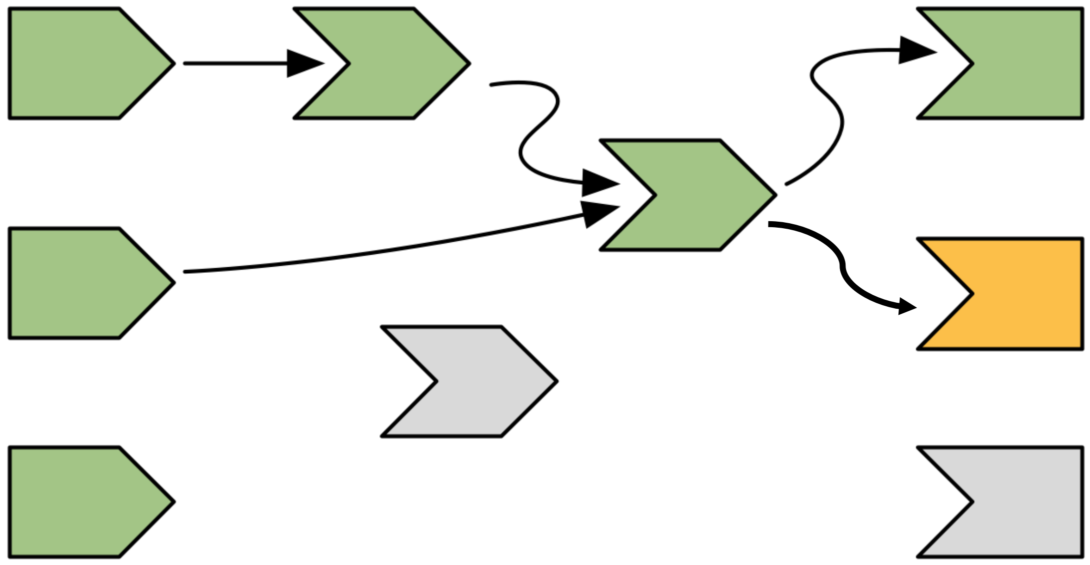


https://shiny.rstudio.com/articles/understanding-reactivity.html

Another way of thinking

- With reactivity, code may not be executed linearly
- We can think of our ‘reactive chain’ from inputs to outputs like a graph
- A reactive expression can go in between, can be a *parent* and/or a *child*

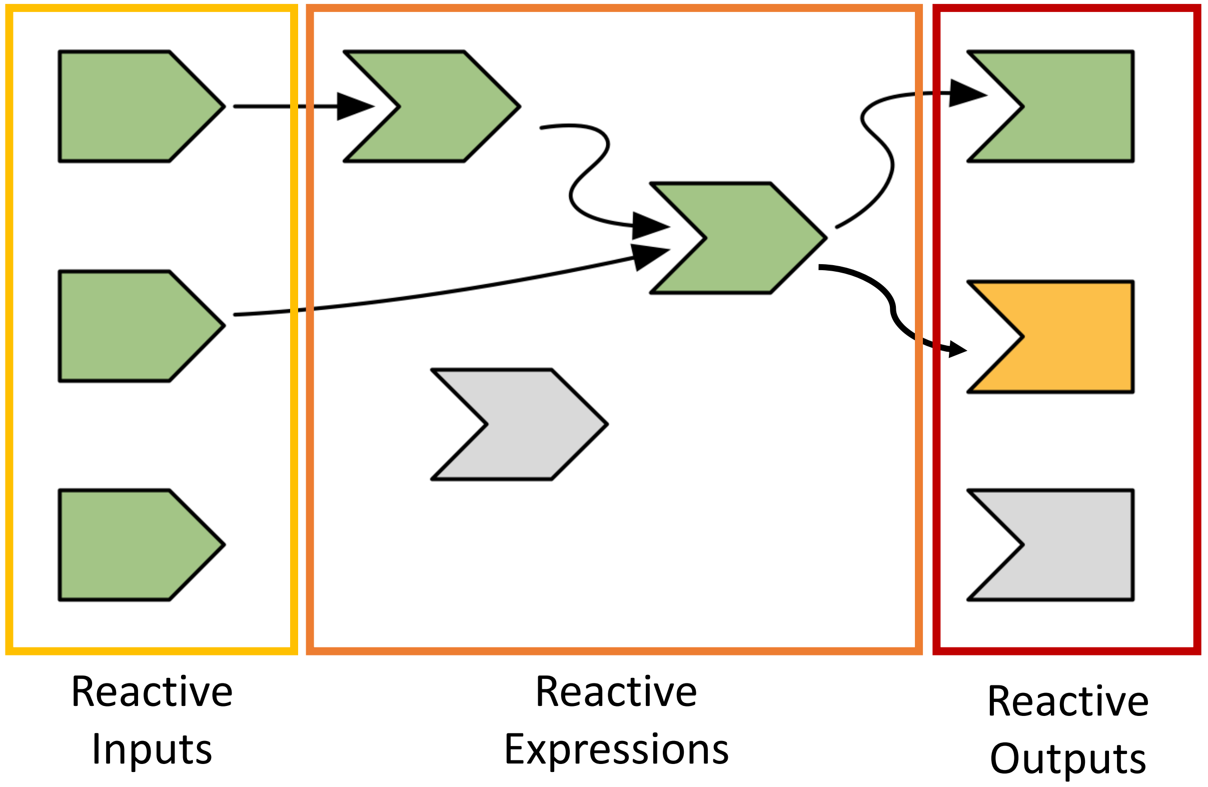
- Let’s look at another example now!



Another way of thinking

- With reactivity, code may not be executed linearly
- We can think of our ‘reactive chain’ from inputs to outputs like a graph
- A reactive expression can go in between, can be a *parent* and/or a *child*

- Let’s look at another example now!



Reactive Context - basics observers

```r
library(shiny)

ui <- fluidPage(
  titlePanel("Reactive values"),
  sidebarLayout(
    sidebarPanel(
      sliderInput('slider','Choose number:',min = 1, max = 20, value =10),
    ),
    mainPanel(
      plotOutput('plot')
    )
  ))
```

Reactive function

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

    toPlot <- reactive({
        return(
          input$slider
        )
    })

    output$plot <- renderPlot(plot(toPlot()))
}
```

Render function

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

    output$plot <- renderPlot(plot(input$slider))

}
```

EPI-interactive

Reactive Context - basic observers

```
library(shiny)

ui <- fluidPage(
  titlePanel("Reactive values"),
  sidebarLayout(
    sidebarPanel(
      sliderInput('slider','Choose number:',min = 1, max = 20, value =10),
    ),
    mainPanel(
      plotOutput('plot')
    )
  ))
```

Reactive function

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

    toPlot <- reactive({
        return(
          input$slider
        )
    })

    output$plot <- renderPlot(plot(toPlot()))
}
```

Render function

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

    output$plot <- renderPlot(plot(input$slider))

}
```

EPI-interactive

Reactive Context – what happens?

Calling a reactive value in a regular function:

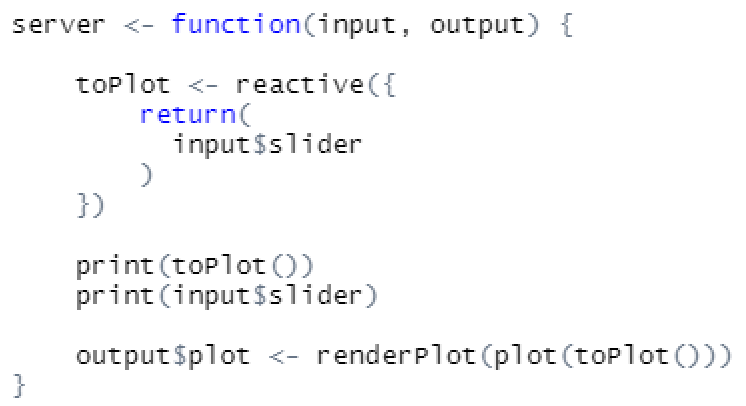
```
library(shiny)

ui <- fluidPage(
  titlePanel("Reactive values"),
  sidebarLayout(
    sidebarPanel(
      sliderInput('slider','Choose number:',min = 1, max = 20, value =10),
    ),
    mainPanel(
      plotOutput('plot')
    )
  ))
```

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

  toPlot <- function(){
      return(
        input$slider
      )
  }

  output$plot <- renderPlot(plot(toPlot()))
}
```

What happens initially and after we change the slider value?

Reactive Context – what happens?

Calling a reactive value by itself in the server:



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

    toPlot <- reactive({
        return(
          input$slider
        )
    })

    print(toPlot())
    print(input$slider)

    output$plot <- renderPlot(plot(toPlot()))
}
```

EPI-interactive

Reactive Input and Output - Exercise

- Either continuing in Session-2/stage1, or starting in /stage2:

- Choose 3 images
  - Sample images are in www folder
  - Name them image_[choice number].jpg
- Assign the choices in your dropdown meaningful names related to the images
- In a reactive function, use if statements to ensure the right image is returned on change of the select input
- Show correct image in mainPanel
- output$display should show the image file name

Reactive Input and Output

server.R

```
server <- function(input, output) {
    #observer
    output$display <- renderText({
        paste(“value= ", selectedImage())
     })
    #reactive value
    selectedImage <- reactive({
        paste0("image_",input$select,".jpg")
    })
    #observer
    output$image <- renderUI({
        img(src = selectedImage(), height = 500)
    })
}
```

ui.R

```
ui <- fluidPage(

    …

    mainPanel(

        uiOutput("image")

    )

)
```

EPI-interactive

Debugging

Debugging in Shiny

It is challenging:

- Reactive, code execution isn’t as linear
- Code runs behind the Shiny framework
- R terminal is busy running the Shiny app

Debugging approaches

- Resetting
- Debugging
- Reprex
- Tracing
- Error handling

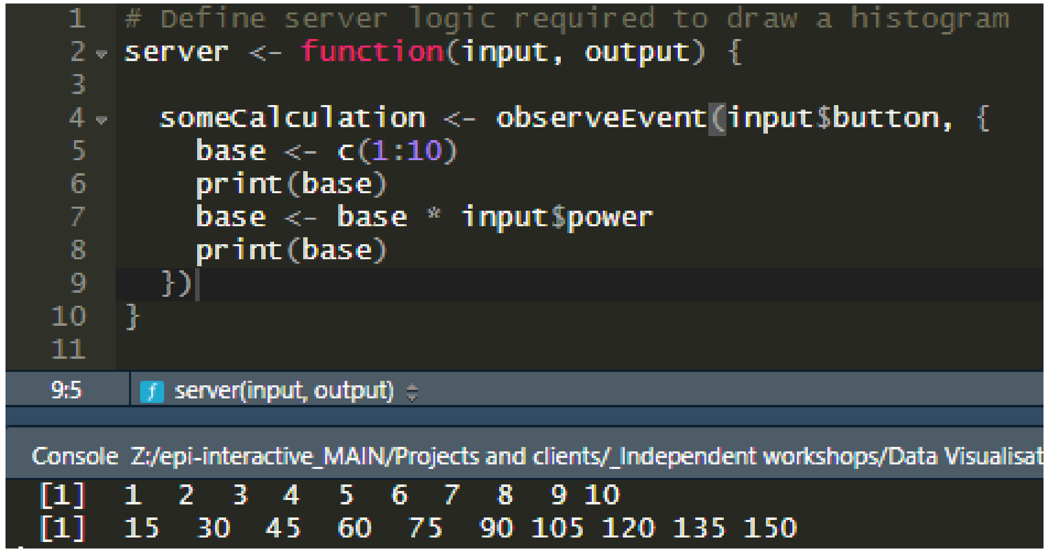
Debugging approaches

- Resetting
- Debugging
- Reprex
- Tracing
- Error handling

Debugging – Reset

- “Have you tried turning it off and on again”
- Need to check if you can reproduce the issue to debug effectively.
- Clear Environment
  - Objects created in global.R or console are stored in the global environment.
  - Clearing environment can prevent issues with left-over variables etc.
  - *R --no-save --no-restore-data*
- Restart R session
  - Can be useful for fixing caching issues (especially theming related)
  - Last resort
- *“Environment should be like Livestock, not house pets”*

Debugging – print()



```
# Define server logic required to draw a histogram
server <- function(input, output) {

  someCalculation <- observeEvent(input$button, {
    base <- c(1:10)
    print(base)
    base <- base * input$power
    print(base)
  })
}

```

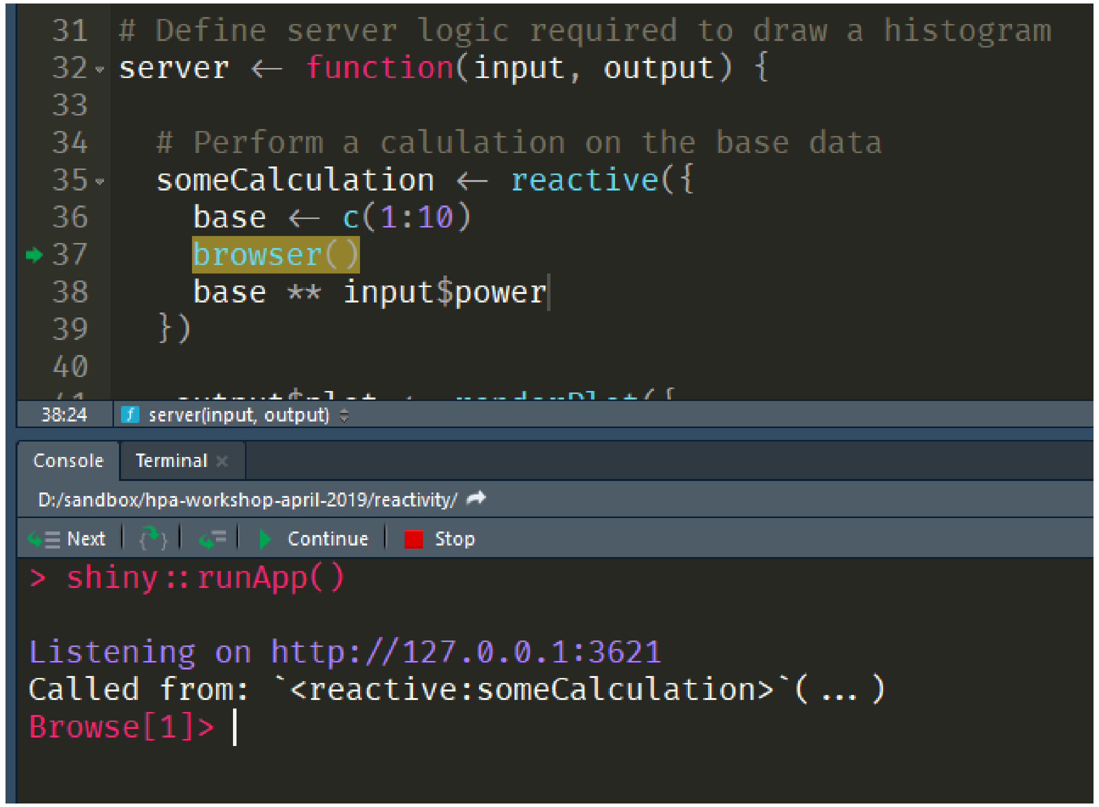
9:5
server(input, output)
Console Z:/epi-interactive_MAIN/Projects and clients/_Independent workshops/Data Visualisat

```
[1]  1  2  3  4  5  6  7  8  9 10
[1]  15  30  45  60  75  90 105 120 135 150
```

- Simple and versatile
- Can check the control flow of an application.
- Can check values during execution
- Good for quick checks

[demo]

Debugging – browser()

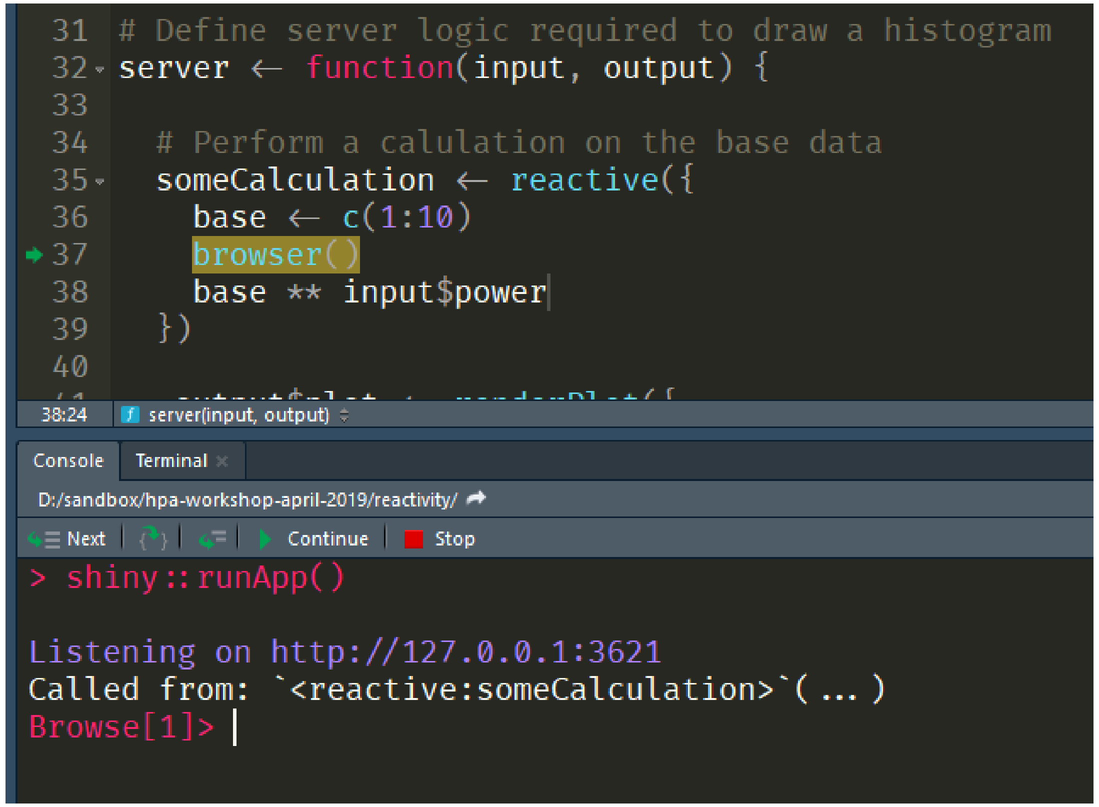


```
# Define server logic required to draw a histogram
server <- function(input, output) {

  # Perform a calulation on the base data
  someCalculation <- reactive({
    base <- c(1:10)
    browser()
    base ** input$power
  })
```

```
> shiny::runApp()

Listening on http://127.0.0.1:3621
Called from: `<reactive:someCalculation>`(...)
Browse[1]>
```

- Works everywhere
- Stops the app and lets us step through each line of code manually
- Great for examining reactive values or for more complex checks

Debugging – browser()



```
# Define server logic required to draw a histogram
server <- function(input, output) {

  # Perform a calulation on the base data
  someCalculation <- reactive({
    base <- c(1:10)
    browser()
    base ** input$power
  })

```

```
> shiny::runApp()

Listening on http://127.0.0.1:3621
Called from: `<reactive:someCalculation>`(...)
Browse[1]>
```

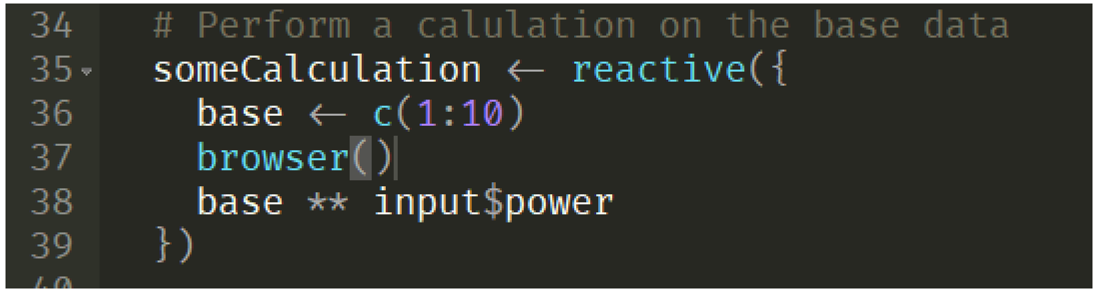
- Add `browser()` in your code
- Run the app and it'll pause when the line is run
- You can step through line by line, enter functions, stop the app, and use the console

Try it out

Using your code from Session 3, or the /result code:

Put **browser()** in the code as below to explore how it works

Type a variable name to see what the value is at that point



```
# Perform a calulation on the base data
someCalculation <- reactive({
  base <- c(1:10)
  browser()
  base ** input$power
})
```

EPI-interactive

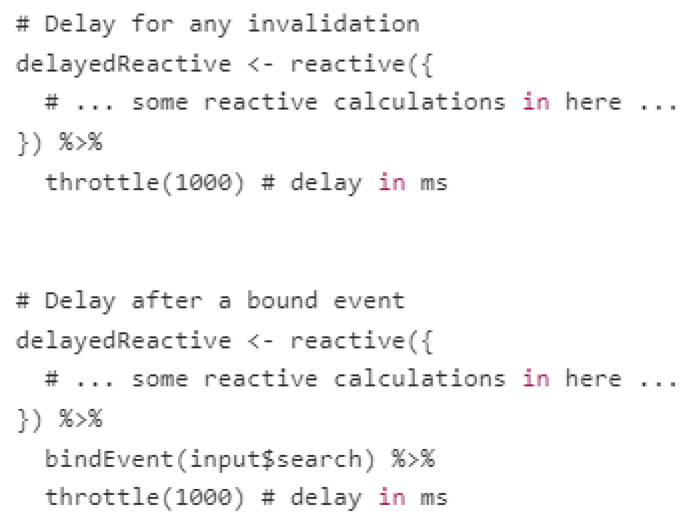
Browser() Summary

- Pauses the code at a certain point
- Very useful for debugging reactive values
- Can be put anywhere
- Lets you step through the code and use the console
- **Important!** Make sure you remove browser() once you are done

EPI-interactive

Reproducable Examples (Reprex)

- Code snippets
- Often used in case of error occurring
- Displayed for simplest case
- Remove unnecessary/excess code



```
# Delay for any invalidation
delayedReactive <- reactive({
  # ... some reactive calculations in here ...
}) %>%
  throttle(1000) # delay in ms


# Delay after a bound event
delayedReactive <- reactive({
  # ... some reactive calculations in here ...
}) %>%
  bindEvent(input$search) %>%
  throttle(1000) # delay in ms
```

EPI-interactive

Next time

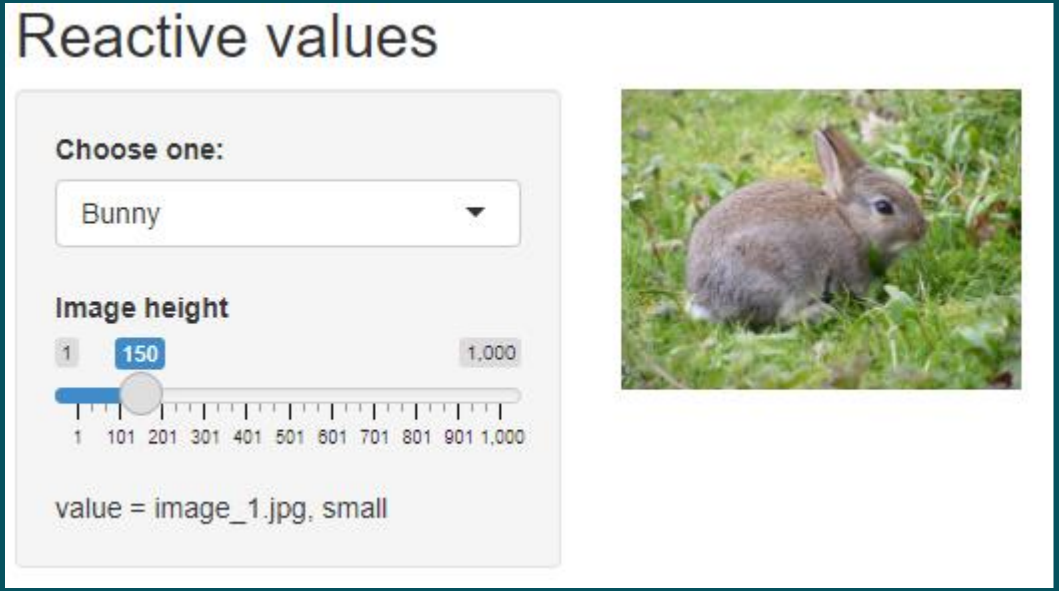
- Data sources
- Data processing

Challenge (using your own /stage2, or /result):

- Add a slider input (min =1, max =1000). This slider will be used to set the height of the image
- Add an option (“None” = 0) to the drop down. When selected, no image should be shown
- Create a reactive function that assigns the slider input value to “small” (<=200), “medium” (201-599) or “large” (600+). If no image is showing, the reactive should return “none”
- output$display should include the result of this new reactive
- Share the link to your project on the **Session 3 forum**

Challenge example

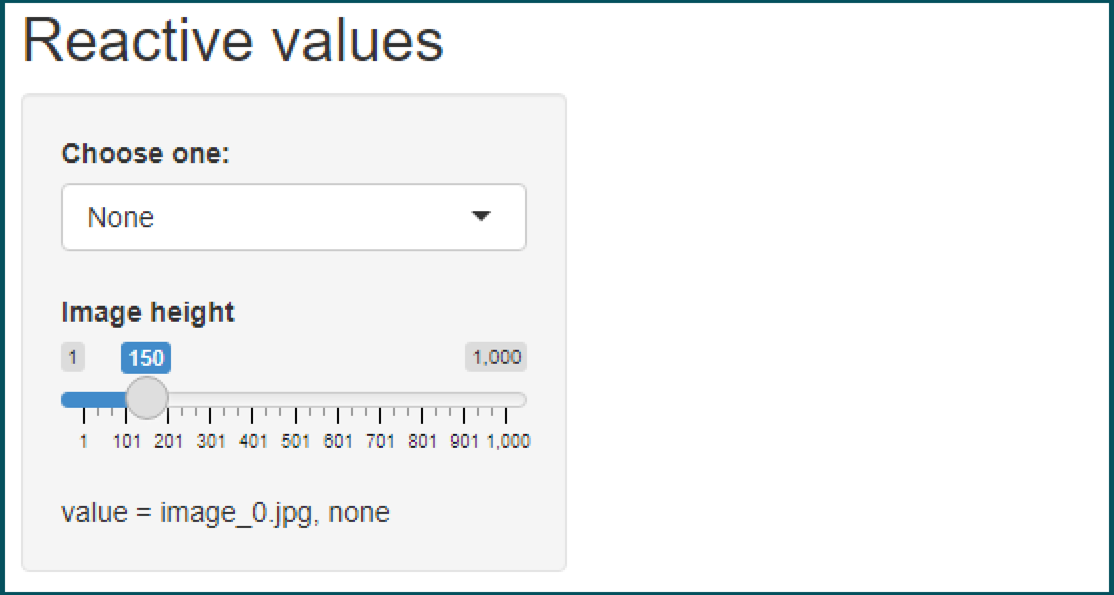
Image showing



Reactive values

value = image_1.jpg, small

No image



Reactive values

value = image_0.jpg, none