October 2024

R Shiny Masterclass Series - Advanced

Session 2

Advanced reactivity

EPI-interactive

Agenda

Session 1 | October 28th | Introduction, recap and responsive interfaces in R Shiny

Session 2 | October 29th | Advanced reactivity

Session 3 | October 31st | Useful R packages to extend core Shiny functionality

Session 4 | November 4th | Managing complexity: modularizing with the module pattern

Session 5 | November 5th | Case Study, Advanced data sources and processing

Session 6 | November 7th | Automated report generation

Session 7 | November 11th | User authentication, Extended exercise

Session 8 | November 12th | AI Tools, Programming sins and how to avoid them

Today

- Continue responsive interface exercise
- Advanced reactivity
  - Observe/observeEvent
  - isolate
  - eventReactive
  - reactiveVal/reactiveValues

Responsive interfaces (ctd.)

renderUI

So far we have been defining our Shiny app UI in the ui.R file…

- Static
- No access to R variables or input values
- No reactivity

Solution: renderUI

- “Modular” approach to defining the UI
- Can improve readability of complex UI layouts

renderUI

- In ui.R, we use **uiOutput("id")**
- In server.R, we use **output$id <- renderUI({ … })**
- Can integrate UI with reactivity, R code and custom data
- Can create input widgets and more uiOutputs inside renderUI

- Final output needs to be a single UI object (div, tagList etc)

renderUI: example

ui.R

```
ui <- bootstrapPage(
  div(style ="padding: 50px",
    fluidRow(
      column(12, selectInput("select", label = "Select
layout", choices = list("C1" = 1, "C2" = 2, "C3" = 3)))
    ),
    fluidRow(
      uiOutput("maincontent")
    ),
    fluidRow(
      uiOutput("calculatedcontent")
    )
  )
)
```

server.R

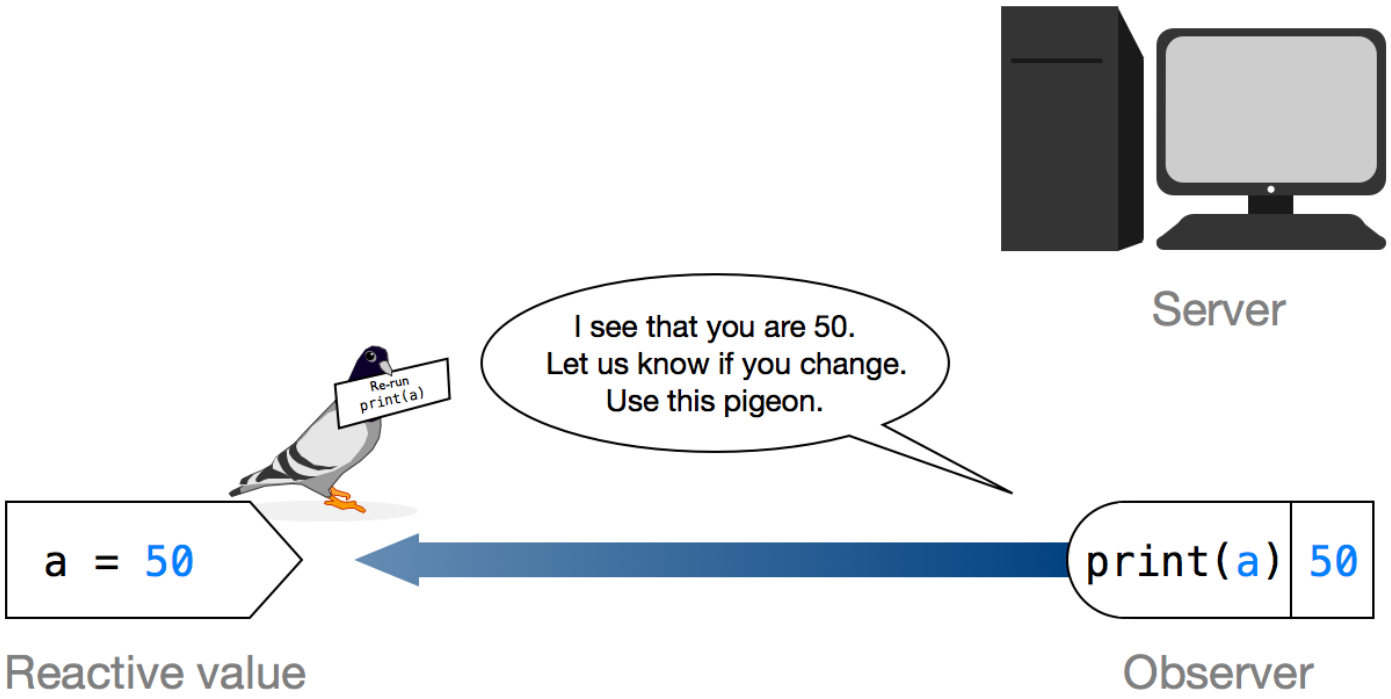
```
server <- function(input, output) {
  output$maincontent <- renderUI({
    column(12, h1(paste0("Layout ", input$select)))
  })

  output$calculatedcontent <- renderUI({
    size <- 12 / as.numeric(input$select)
    tagList(
      lapply(X = c(1:input$select), FUN = function(x, size) {
        column(size, wellPanel(paste0("Column ", x)))
      }, size)
    )
  })
}
```

How will this look in a Shiny app?
What does this code do?

Advanced Reactivity

Reactivity Explained



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

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")
    )
)
```

Reactive Context

- Where can we safely use reactive values and expressions
- Reactive expressions and values only exist while the application is running
- Separate for each individual user
- ‘Reactive Context’ is areas in the code that only exist at runtime

server.R

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

    .#........ #.
    .# not here #.
    .#........ #.

    # inside these expressions is fine
    output$display <- renderText({
        paste(“value= ", selectedImage())
     })
    selectedImage <- reactive({
        paste0("image_",input$select,".jpg")
    })
    output$download <- downloadHandler(
    filename = function(){ … },
    content = function(file) { … }
    })
}
```

Beyond reactive()

- observe/observeEvent: watch for changes
- isolate: stop reactions
- eventReactive: calculate expressions based on events
- reactiveValues/reactiveVal: reactive variables
- Update inputs: automatically update inputs

Evaluation methods

- **Eager**
  - Triggers by itself when the observed values change. Does not need to be called separately.
  - Example: observe, observeEvent
- **Lazy**
  - In order to trigger, it needs to be called in a reactive context.
  - Example: reactive, eventReactive, reactiveValues*, reactiveVal*

Eager evaluation

Observe vs ObserveEvent: Syntax

```
observe({ x })
```

- **x:** what to do once any of the values you are watching have changed (defined in x)

```
observeEvent(eventExpr, { handlerExpr })
```

- **eventExpr**: The reactive value(s) you are watching
- **handlerExpr:** what to do once eventExpr has changed

Isolate

`Isolate(expr)`

- Expr: An expression that can access reactive values or expressions

- Is used to stop a reaction
- Will only take the current value, will not trigger other outer reactive contexts
- Example: in observe()

Comparison: Eager

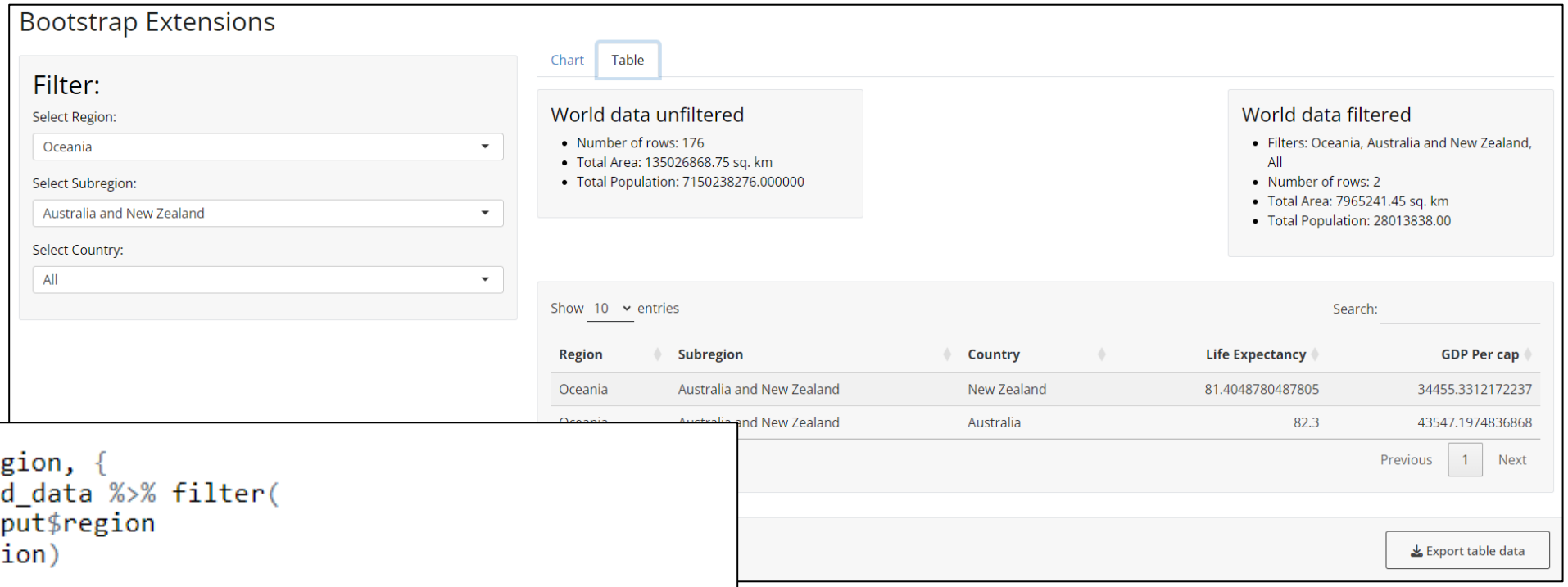
Feature	observe	observeEvent
Expects a return value	✗	✗
Able to watch many values	✓	✓
Contains an isolated scope	✗	✓
Reactive context	✓	✓
Can be reassigned	✗	✗
Create dependents*	✗	✗
Can execute code**	✓	✓

**Dependents are reactive values that can trigger changes with reactive contexts*
***Rather than just containing values*

Observe, observeEvent and isolate

1. A button is pressed, and we only want to calculate something after the button is pressed.
2. We want to display a pop-up as soon as some reactive calculation completes, displaying default values of inputs x, y and z. However, we don’t want to display this popup again if any of the values change later.
3. We want to run some code using input x and input y any time either of them change.
4. We want to use the value of a text input in some reactive calculation, but not have this trigger reactive behaviour

Example




```
observeEvent(input$region, {
  sub_regions <- world_data %>% filter(
    region_un %in% input$region
  ) %>% select(subregion)

  updateSelectInput(inputId="subregion",
                    choices=c("All", unique(sub_regions$subregion))
  )
})
```

Lazy evaluation

reactive()

- A ‘reactive expression’
- Contains a block of standard R code which can use reactive values
- R code will execute when any reactive values in the expression update
- Last result is cached behind the scenes for quick retrieval

```
# defining a reactive
Example <- reactive({
# this code will run
whenever some reactive
value changes inside
this code block

return(input$num * 2)
})

# using a reactive
output$text <-
renderText(Example())
```

eventReactive

```
varA <- eventReactive(eventExpr, {valueExpr})
```

- **eventExpr:** The event/reactive value you are watching
- **valueExpr:** The expression that produces the return value of the eventReactive

- Used to calculate a reactive expression in response to an event or events
- Called like a normal reactive function (`varA()`)
- The expression is in an isolated scope
- Returns NULL until the first time the expression runs

reactiveVal

```
value <- reactiveVal()
```

- Optional parameters value and label

- One value at a time
- Get: `reactiveVal()`
- Set: `reactiveVal([new value])`

reactiveValues

- A ‘reactive list’
- values <- reactiveValues(...)
- Function returns an object for storing reactive values
- Each value must have a name

Get:

```
values #get all
values$a / values[[‘a’]] #get one
```

Set:

```
values <- reactiveValues(a = 1, b = 2) #set all
values$a <- 3 #set one
values[['b']] <- 4 #set one
```

ReactiveVal vs ReactiveValues

```
value <- reactiveVal(0)

observeEvent(input$minus,
{
newValue <- value() - 1

value(newValue)
})
```

```
rv <- reactiveValues(
value = 0)

newValue <- rv$value - 1

rv$value <- newValue
```

Comparison: Lazy

Feature	reactive()	eventReactive	reactiveValues / reactiveVal
Expects a return value	✓	✓	✗
Able to watch many values	✓	✓	✗
Contains an isolated scope	✗	✓	✗
Reactive context	✓	✓	✗
Can be reassigned in place	✗	✗	✓
Create dependents*	✓	✓	✓
Can execute code**	✓	✓	✗

**Dependents are reactive values that can trigger changes with reactive contexts*
***Rather than just containing values*

bindEvent

```
varA <- eventReactive(eventExpr, {valueExpr})
varB <- reactive({valueExpr}) %>% bindEvent(eventExpr)
```

- **eventExpr:** The event/reactive value you are watching
- **valueExpr:** The expression that produces the return value of the eventReactive

- Equivalent to using observeEvent or eventReactive (scope wise)
- Can also be used with shiny render functions (renderUI, renderPlot, etc).
- Works well with bindCache (more on that later).

Exercise

Using **/stage1:**

- Keep track of how many times each animal button is pressed, using the ‘votes’ reactiveValues object
- In the last empty wellPanel, create a row and two columns. In these columns:
  - Create a renderUI called “liveDisplay” and use this to show the current votes for each animal
  - Create a renderUI called “finalDisplay”, which should only be visible once the user clicks “Reveal votes”. This display should only update when the ‘Reveal votes’ button is clicked.
- Reset the vote counts if the user hits “Reset”

Use a combination of reactiveValues / reactiveVal, eventReactive, observeEvent and renderUI to achieve this - the implementation is up to you!

Exercise example

Vote for your favourite!

Sea lion | Puffin | Lamb | Horse

Reveal votes | Reset

Live results
sealion: 1
puffin: 3
lamb: 2
horse: 1

Final results
sealion: 1
puffin: 3
lamb: 2
horse: 1

Update inputs

- To automatically change the input values (without user interaction)
- Generally because of another action (eg button click or change of another input)
- update[input type](session, inputId, …)
  - NOTE: inputId is “id” not input$id
  - “…” is the parameters you want to change, you do **not** have to put all parameters
- Examples: https://shiny.rstudio.com/gallery/update-input-demo.html

Next time

- Reactivity Practice
- UX considerations
- Useful R packages to extend core Shiny functionality
  - DT
  - shinyJS
  - shiny.router

Challenge

Using your exercise code or the /challenge folder:

- Order the final results, should be in order from most to least, in “finalDisplay”
- After votes are cast and the user selects “Reveal votes” only display the winning picture. Use a reactiveVal to keep track of the winner.
- If the user hits reset, all the images should display, and the winner should be reset
- If there is more than one winner, show a message, using validate, where the images would display
- Share your work on the Session 2 forum!

Challenge example