REACTIVE PROGRAMMING
PART 2

Shiny from RStudio™

Stop - trigger - delay

Stop with isolate()

ISOLATE

‣ Use `isolate()` to wrap an expression whose reactivity should be suppressed (i.e. the currently executing reactive expression/observer/output shouldn't be notified when something changes).

DEMO

Only update plot title when other components of the plot are also updated. See `movies_14.R`.

server:

```
pretty_plot_title <- reactive({ toTitleCase(input$plot_title) })

output$scatterplot <- renderPlot({
  ggplot(data = movies_subset(), aes_string(x = input$x, y = input$y, color = input$z)) +
    geom_point(alpha = input$alpha, size = input$size) +
    labs(title = isolate({ pretty_plot_title() }) )
})
```

Plot title will update when any of the other **input**s in this chunk change

Plot title will **not** update when **input$plot_title** changes

Trigger with observeEvent()

TRIGGERING A REACTION

‣ observeEvent() can be used to trigger a reaction

‣ It uses a different syntax

observeEvent(eventExpr, handlerExpr, …)

simple reactive value - input$click,
call to reactive expression - df(),
or complex expression inside {}

expression to call whenever
eventExpr is invalidated

Shiny from R Studio™

`</>`

Write a CSV of the sampled data when action button is pressed. See `movies_15.R`.

ui:

```
actionButton(inputId = "write_csv", label = "Write CSV")
```

server:

```
observeEvent(eventExpr = input$write_csv,
            handlerExpr = {
              filename <- paste0("movies_", str_replace_all(Sys.time(), ":|\ ", "_"), ".csv")
              write.csv(movies_sample(), file = filename, row.names = FALSE)
              }
            )
```

Shiny from **R** Studio™

ISOLATE VS. OBSERVEEVENT

- `isolate()` is used to stop a reaction

- while `observeEvent()` is used to perform an action in response to an event

  - Note: "recalculate a value" does not generally count as performing an action, we'll next discuss `eventReactive()` for that

Delay reactions with eventReactive()

OBSERVEEVENT VS. EVENTREACTIVE

‣ observeEvent() is to to perform an action in response to an event

‣ while eventReactive() is used to create a calculated value that only updates in response to an event

  ‣ Just like a normal reactive expression except only invalidates in response to the given event.

```
observeEvent(eventExpr, valueExpr, …)
```

‣ Change how the random sample is generated such that it is updated when the user clicks on an action button that says "Get new sample".

‣ Use `movies_15.R` as the basis of the script and make the updates there.

‣ Run the app to ensure that the behavior is as described

‣ Compare your code / output with the person sitting next to / nearby you

5m 00s

Solution can also be found in `movies_16.R`.

ui:

```r
actionButton(inputId = "get_new_sample",
             label = "Get new sample")
```

server:

```r
movies_sample <- eventReactive(eventExpr = input$get_new_sample,
                               valueExpr = {
                                 req(input$n_samp)
                                 sample_n(movies_subset(), inpu
                               },
                               ignoreNULL = FALSE
)
```

Initially perform the action/ calculation and just let the user re-initiate it (like a "Recalculate" button)

Scheduling

Schedule with invalidateLater()

INVALIDATELATER

‣ If this is placed within an observer or reactive expression, that object will be invalidated (and re-execute) after the interval has passed

‣ The re-execution will reset the invalidation flag, so in a typical use case, the object will keep re-executing and waiting for the specified interval.

‣ It's possible to stop this cycle by adding conditional logic that prevents the `invalidateLater()` from being run.

Tell the user how long they have been viewing your app for. See `movies_17.R`.

ui:

```
textOutput(outputId = "time_elapsed")
```

server:

```
# Calculate time difference between when app is first launched and now
beg <- reactive({ Sys.time() })
now <- reactive({ invalidateLater(millis = 1000); Sys.time() })
diff <- reactive({ round(difftime(now(), beg(), units = "secs")) })

# Print time viewing app
output$time_elapsed <- renderText({
  paste("You have been viewing this app for", diff(), "seconds.")
})
```

Shiny from RStudio™

- ‣ Change how the random sample is generated such that it is updated every 5 seconds

  - ‣ Don't forget to remove now unused functionality for the action button to get a new sample

- ‣ Use `movies_17.R` as the basis of the script and make the updates there

- ‣ Run the app to ensure that the behavior is as described

- ‣ Compare your code / output with the person sitting next to / nearby you

5ₘ 00ₛ

SOLUTION

Solution can also be found in `movies_18.R`.

ui:

```r
actionButton(inputId = "get_new_sample", label = "Get new sample")
```

server:

```r
# Get new sample every 5 seconds
movies_sample <- reactive({ invalidateLater(millis = 5000)
    req(input$n_samp)
    sample_n(movies_subset(), input$n_samp)
  })
```

Shiny from RStudio™

Monitor with reactivePoll()

- `reactivePoll()` pairs a relatively cheap "check" function with a more expensive value retrieval function

  - Check function: is executed periodically and should always return a consistent value until the data changes

    - Note doesn't return TRUE or FALSE, instead it indicates change by returning a different value from the previous time it was called

  - Value retrieval function: is used to re-populate the data when the check function returns a different value

- Similar to `invalidateLater()`, but it's based on a change in a file as opposed to a periodic change

Periodically check and report the names and dimensions of CSV files in the directory.

1. Write the check and value retrieval functions for `reactivePoll()`
2. Count and list CSV files in the directory every 5 seconds with `reactivePoll()`
3. Store CSV files in the directory as a data table in `output$csv_files`
4. Print `output$csv_files` in the UI, use tabs to reduce clutter

1. Write the check and value retrieval functions for `reactivePoll()`

```r
# Check function
count_files <- function(){ length(dir(pattern = "*.csv")) }

# Value retrieval function
list_files <- function(){
  files <- dir(pattern = "*.csv")
  if(length(files) == 0){ return( data.frame() ) }
  sapply(files, function(file) dim(read.csv(file))) %>%
    unlist() %>%
    t() %>%
    as.data.frame() %>%
    setNames(c("rows", "cols"))
}
```

There are many ways of doing this, don't focus too much on this code

2. Count and list CSV files in the directory every 5 seconds with
   `reactivePoll()`

```r
# Count and list CSV files in the directory every 5 seconds
csv_files <- reactivePoll(intervalMillis = 5000,
             session,
             checkFunc = count_files,
             valueFunc = list_files)
```

3. Store CSV files in the directory as a data table in output$csv_files

```r
# Print CSV files in the directory
output$csv_files <- DT::renderDataTable(
    DT::datatable(data = csv_files(),
                  options = list(pageLength = 10),
                  rownames = TRUE)
```

Shiny from RStudio™

4. Print output$csv_files in the UI, use tabs to reduce clutter

```r
# Use tabs for the data tables to reduce clutter
tabsetPanel(
  # Show data table
  tabPanel("Plotted data", dataTableOutput(outputId = "moviestable")),

  # Show CSV files in directory
  tabPanel("Files in directory", dataTableOutput(outputId = "csv_files"))
)
```

This is new syntax we haven't seen before

Putting it all together…

`movies_19.R`

See it in action: Change sample size, get new sample,
write data to CSV, check out the "Files in directory" tab.
Then, delete all CSV files in directory, and see the list update.

reactiveFileReader()

REACTIVEFILEREADER

‣ `reactiveFileReader()` works by periodically checking the file's last modified time

 ‣ If the file has changed, it is re-read and any reactive dependents are invalidated

‣ Also similar to `invalidateLater()` but instead of periodic updates, updates are based on changes in a file

Reactivity
best practices

Is there something wrong with this? If so, what?

```r
ui <- fluidPage(
  titlePanel("Add 2"),
  sidebarLayout(
    sidebarPanel( sliderInput("x", "Select x", min = 1, max = 50, value = 30) ),
    mainPanel( textOutput("x_updated") )
  )
)

server <- function(input, output) {
  add_2            <- function(x) { x + 2 }
  current_x        <- add_2(input$x)
  output$x_updated <- renderText({ current_x })
}
```

1m 00s

Yup! See add_2.R.

```r
ui <- fluidPage(
  titlePanel("Add 2"),
  sidebarLayout(
    sidebarPanel( sliderInput("x", "Select x", min = 1, max = 50, value = 30) ),
    mainPanel( textOutput("x_updated") )
  )
)

server <- function(input, output) {
  add_2            <- function(x) { x + 2 }
  current_x        <- reactive({ add_2(input$x) })
  output$x_updated <- renderText({ current_x() })
}
```

Shiny from RStudio™

Reactives are equivalent to no argument functions

Think about them as functions, think about them as variables that can depend on user input and other reactives

observe() vs. reactive()

Which one should you use if you want to create an object that you can later use in a render function?

Which one if you want to update the minimum value of a slider input based on the choices a user makes in the app?

1m 00s

observe() vs. reactive()

Which one should you use if you want to create an object that you can later use in a render function?

reactive()

Which one if you want to update the minimum value of a slider input based on the choices a user makes in the app?

observe()

Shiny from RStudio

Reactives are for reactive values and expressions

Observers are for their side effects

Is there something wrong with this? If so, what?

```r
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(sliderInput("n", "Select n", min = 1,
                              max = 50, value = 30)),

    mainPanel(
      plotOutput("hist"),
      textOutput("med")
    )
  )
)
```

```r
server <- function(input, output) {
  dist <- reactive({ rnorm(input$n) })
  output$hist <- renderPlot({
    hist(dist())
    med <- reactive({ median(dist()) })
    abline(v = med(), col = "red")
  })
  output$med <- renderText({
    paste("The median is", round(med(), 3))
  })
}
```

1ₘ **00**ₛ

Oh yeah! See `hist_med.R`.

```r
ui <- fluidPage(
 sidebarLayout(
   sidebarPanel(sliderInput("n", "Select n", min = 1,
                            max = 50, value = 30)),
   mainPanel(
     plotOutput("hist"),
     textOutput("medtext")
   )
 )
)
```

```r
server <- function(input, outPut) {
  dist <- reactive({ rnorm(input$n) })
  med <- reactive({ median(dist()) })
  output$hist <- renderPlot({
    hist(dist())
    abline(v = med(), col = "red")
  })
  output$medtext <- renderText({
    paste("The median is", round(med(), 3))
  })
}
```

Shiny from **R**Studio™